HOEPLI
TECNICA
PER LA SCUOLA

PAOLO CAMAGNI
RICCARDO NIKOLASSY

TECNOLOGIE E PROGETTAZIONE DI SISTEMI INFORMATICI E DI TELECOMUNICAZIONI

Per l'articolazione **INFORMATICA**
degli Istituti Tecnici
settore Tecnologico

2

RISORSE ONLINE

HOEPLI

Tecnologie e progettazione di sistemi informatici e di telecomunicazioni

PAOLO CAMAGNI RICCARDO NIKOLASSY

Tecnologie e progettazione di sistemi informatici e di telecomunicazioni

Per l'articolazione Informatica degli Istituti Tecnici settore Tecnologico

VOLUME 2

EDITORE ULRICO HOEPLI MILANO

UN TESTO PIÙ RICCO E SEMPRE AGGIORNATO
Nel sito **www.hoepliscuola.it** sono disponibili:
- materiali didattici integrativi;
- eventuali **aggiornamenti** dei contenuti del testo.

Via Hoepli 5, 20121 Milano (Italy)
tel. +39 02 864871 – fax +39 02 8052886
e-mail hoepli@hoepli.it

www.hoepli.it

Indice

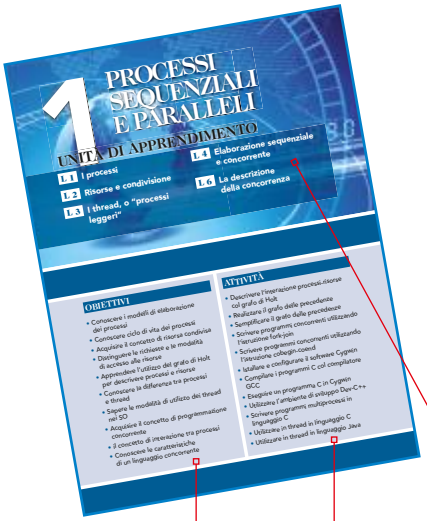
Presentazione

L'impostazione del presente volume fa riferimento agli obiettivi specifici di apprendimento proposti dal ministero per la nuova disciplina Tecnologie e Progettazione di Sistemi Informatici e di Telecomunicazioni introdotta dalla riforma nel corso di informatica per il nuovo triennio della scuola superiore.

Abbiamo ritenuto irrinunciabile fare tesoro della nostra esperienza maturata nel corso di numerosi anni di insegnamento che ci ha reso consapevoli della difficoltà di adeguare le metodologie didattiche alle dinamiche dell'apprendimento giovanile e ai continui cambiamenti tecnologici che implicano sempre nuove metodologie di comunicazione, per proporre un testo con una struttura innovativa, riducendo l'aspetto teorico e proponendo un approccio didattico di apprendimento operativo, privilegiando il "saper fare".

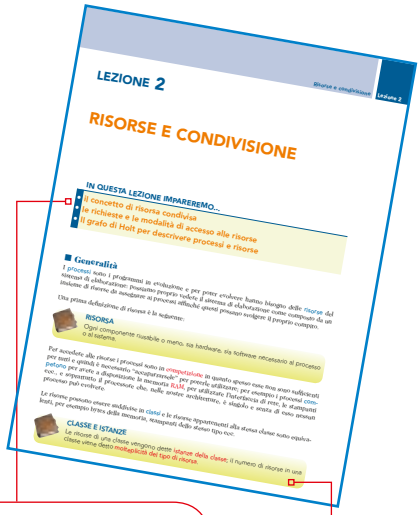
Il volume è strutturato in quattro unità di apprendimento suddivisi in lezioni che ricalcano le indicazioni dei programmi ministeriali per il quarto anno di studio: lo scopo di ciascuna unità di apprendimento è quello di presentare un intero argomento, mentre quello delle lezioni è quello di esporne un singolo aspetto.

Le finalità e i contenuti dei diversi argomenti affrontati sono presentati all'inizio di ogni unità di apprendimento; in conclusione di ogni lezione sono presenti esercizi di valutazione delle conoscenze e delle competenze raggiunte, suddivisi in domande a risposta multipla, vero o falso, a completamento, e nelle lezioni specifiche per il laboratorio esercizi di progettazione da svolgere autonomamente o in gruppo di lavoro.



Nella pagina iniziale di ogni unità di apprendimento è presente un indice delle lezioni trattate

Indice degli obiettivi che si intendono raggiungere e delle attività che si sarà in grado di svolgere



All'inizio di ogni lezione sono indicati in modo sintetico i contenuti

Il significato di moltissimi termini informatici viene illustrato e approfondito

L'argomento fondamentale, cioè la programmazione concorrente, è stato affrontato e descritto sia per la codifica in linguaggio Java che in linguaggio C, in modo da offrire una visione globale e comparativa delle due diverse tecnologie.

In questa sezione viene approfondito un argomento di particolare importanza

Le osservazioni aiutano lo studente a comprendere e ad approfondire

Le parole chiave vengono poste in evidenza e spiegate allo studente

RISORSE ONLINE

Alla pagina web **http://www.hoepliscuola.it** sono disponibili le risorse online, tra cui unità didattiche integrative, numerosi esercizi aggiuntivi per il recupero e il rinforzo, nonché schede di valutazione di fine unità.

Per la verifica delle conoscenze e delle competenze è presente un'ampia sezione di esercizi

1 PROCESSI SEQUENZIALI E PARALLELI

UNITÀ DI APPRENDIMENTO

L 1 I processi

L 2 Risorse e condivisione

L 3 I thread, o "processi leggeri"

L 4 Elaborazione sequenziale e concorrente

L 6 La descrizione della concorrenza

OBIETTIVI

- Conoscere i modelli di elaborazione dei processi
- Conoscere ciclo di vita dei processi
- Acquisire il concetto di risorsa condivisa
- Distinguere le richieste e le modalità di accesso alle risorse
- Apprendere l'utilizzo del grafo di Holt per descrivere processi e risorse
- Conoscere la differenza tra processi e thread
- Sapere le modalità di utilizzo dei thread nei SO
- Acquisire il concetto di programmazione concorrente
- il concetto di interazione tra processi
- Conoscere le caratteristiche di un linguaggio concorrente

ATTIVITÀ

- Descrivere l'interazione processi-risorse col grafo di Holt
- Realizzare il grafo delle precedenze
- Semplificare il grafo delle precedenze
- Scrivere programmi concorrenti utilizzando l'istruzione fork-join
- Scrivere programmi concorrenti utilizzando l'istruzione cobegin-coend
- Istallare e configurare il software Cygwin
- Compilare i programmi C col compilatore GCC
- Eseguire un programma C in Cygwin
- Utilizzare l'ambiente di sviluppo Dev-C++
- Scrivere programmi multiprocessi in linguaggio C
- Utilizzare in thread in linguaggio C
- Utilizzare in thread in linguaggio Java

LEZIONE 1

I PROCESSI

IN QUESTA LEZIONE IMPAREREMO...

- i modelli di elaborazione dei processi
- Il ciclo di vita dei processi

Il modello a processi

Nonostante l'evoluzione tecnologica abbia incrementato le capacità computazionali grazie all'aumento della velocità delle CPU, la gestione del processore ancora oggi deve essere ottimizzata perché spesso presenta situazioni di criticità: tutti i moderni SO cercano di sfruttare al massimo le potenzialità di parallelismo fisico dell'hardware per minimizzare i tempi di risposta e aumentare il throughput del sistema, ossia il *numero di programmi eseguiti per unità di tempo*.

Il programma, composto da un insieme di byte contenente le istruzioni che dovranno essere eseguite, è un'entità passiva finché non viene caricato in memoria e mandato in esecuzione: diviene un processo che evolve man mano che le istruzioni vengono eseguite dalla CPU, cioè si trasforma in un'entità attiva,

Possiamo dare come definizione sintetica di processo quella riportata di seguito.

PROCESSO

Un processo è un'entità logica in evoluzione.

Nel modello a processi tutto il software che può essere eseguito su di un calcolatore, compreso il Sistema Operativo stesso, è organizzato in un certo numero di processi sequenziali (successivamente chiamati semplicemente processi): un unico processore può essere condiviso tra parecchi processi, utilizzando un algoritmo di schedulazione (di scheduling) per determinare quando interrompere l'evoluzione di un processo e servirne un altro.

Questa tecnica di gestione della CPU si chiama multiprogrammazione, che richiede la contemporanea presenza di più programmi in memoria: dato che l'esecuzione di un programma, quindi

un processo, è costituita da una successione di fasi di elaborazione della CPU e fasi di attesa per l'esecuzione di operazioni su altre risorse del sistema (operazioni di I/O, di caricamento dati, di colloquio con periferiche ecc.) che di fatto lasciano inattiva la CPU, la multiprogrammazione permette l'evoluzione contemporanea di più processi limitando al minimo i tempi morti e sfruttando appieno le potenzialità di calcolo del processore.

È anche possibile avere in esecuzione contemporaneamente più istanze di un programma, quindi più processi originati dallo stesso codice: per esempio si possono avere aperte due finestre con lo stesso programma in esecuzione.

Inoltre i processi possono essere indipendenti oppure cooperare per raggiungere un medesimo obiettivo:

- nel primo caso, cioè di processi indipendenti, un processo evolve in modo autonomo senza bisogno di comunicare con gli altri processi per scambiare dati;
- nel secondo caso, due (o più) processi hanno la necessità di cooperare in quanto, per poter evolvere, necessitano di scambiarsi informazioni. Si pensi per esempio a un semplice videogame con due giocatori dove ogni giocatore è un processo: in questo caso nasce la necessità per i due processi di coordinarsi per poter comunicare e scambiarsi le informazioni (i processi si devono sincronizzare).

Oltre alla cooperazione esiste un'altra forma di interazione tra processi: due (o più) processi possono ostacolarsi a vicenda compromettendo il buon fine delle loro elaborazioni.
È il caso in cui entrambi i processi competono per utilizzare la medesima risorsa, che magari è in quantità limitata nel sistema: questo tipo di interazione può portare a situazioni indesiderate per uno o per entrambi i processi (blocco individuale o critico).

Riassumendo, abbiamo quindi tre modelli di computazione per i processi:

- modello di computazione indipendente;
- modello di computazione con cooperazione;
- modello di computazione con competizione.

Diamo una definizione per le due ultime situazioni:

PROCESSI COOPERANTI

Un processo è cooperante se influenza o può essere influenzato da altri processi in esecuzione nel sistema (un processo che condivide dati con altri processi).

La possibilità di avere processi che cooperano, in sintesi, è utile per ottenere:

- **parallelizzazione dell'esecuzione** (per esempio macchine con multi-cpu);
- **replicazione di un servizio** (per esempio connessioni di rete);
- **modularità** (diversi processi per funzioni diverse di una stessa applicazione, come per esempio il correttore ortografico di Word (posso continuare a scrivere mentre corregge, cioè posso compiere più azioni contemporaneamente).
- **condivisione delle informazioni.**

PROCESSI IN COMPETIZIONE

Due processi sono in competizione se possono evolvere indipendentemente ma entrano in **conflitto sulla ripartizione delle risorse**.

L'esempio più semplice di competizione nella multiprogrammazione è dato dallo scheduling dei processi, dove tutti *competono per la CPU*; un'altra risorsa che è sempre **condivisa** è la stampante, e per accedervi i processi devono attendere in coda (*competono per la risorsa stampante*).

In questo volume analizzeremo le possibili forme di interazione, desiderate e indesiderate, e affronteremo il progetto di applicazioni concorrenti scrivendone la codifica in linguaggio di programmazione.

Stato dei processi

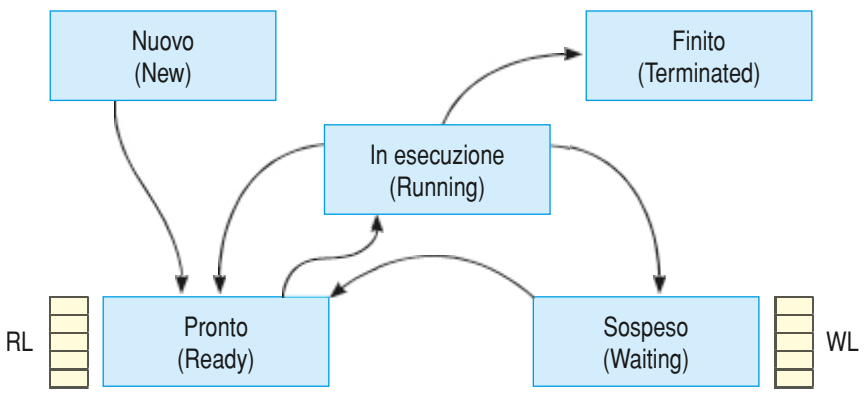
Durante il ciclo di vita di un processo è possibile individuare un insieme di situazioni in cui il processo può trovarsi, che definiremo come gli stati di un processo, associati alla sua evoluzione e alla sua "situazione" rispetto alla CPU.
Vediamo dettagliatamente come può trovarsi un processo rispetto al processore:

- nuovo (*new*): è lo stato in cui si trova un processo appena è stato creato, cioè l'utente richiede l'esecuzione di un programma che risiede sul disco;
- esecuzione (*running*): il processo sta evolvendo, nel senso che la CPU sta eseguendo le sue istruzioni, e quindi a esso è assegnato il processore. Nei sistemi a monoprocessore (quelli analizzati in questo testo) un solo processo può essere in questo stato;
- attesa (*waiting*): un processo è nello stato di attesa quando gli manca una risorsa per poter evolvere (oltre alla CPU) e quindi sta *aspettando* che si verifichi un evento (per esempio che si liberi la risorsa che gli serve e che il processore gliela assegni);
- pronto (*ready-to-run*): un processo è nello stato di pronto se ha tutte le risorse necessarie alla sua evoluzione tranne la CPU (cioè è il caso in cui sta aspettando che gli venga assegnato il suo time-slice di CPU);
- finito (*terminated*): siamo nella situazione in cui tutto il codice del processo è stato eseguito e quindi ha determinato l'esecuzione; il sistema operativo deve ora rilasciare le risorse che utilizzava.

STATO DI UN PROCESSO

Con stato di un processo intendiamo quindi una tra le cinque possibili situazioni in cui un processo in esecuzione può trovarsi: può assumere una sola volta lo stato di nuovo e di terminato, mentre può essere per più volte negli altri tre stati.

Vediamo come è possibile rappresentare mediante un grafico orientato i passaggi che un processo esegue tra i possibili stati sopra descritti: questo grafico prende il nome di diagramma degli stati.



Seguiamo ora la vita di un processo dal principio: al nuovo processo viene assegnato un identificatore (PID, Process IDentifier) e viene inserito nell'elenco dei processi pronti (RL, Ready List) in attesa che arrivi il suo turno di utilizzo della CPU.

> Ogni processo è creato da un altro processo detto "padre" mentre lui prende il nome di "figlio": l'unica eccezione è il processo "init", cioè il primo a essere eseguito dal S.O, che non ha nessun "padre".

Quando gli viene assegnata la CPU, il processo passa nello stato di esecuzione, dal quale può uscire per tre motivi:

- termina la sua esecuzione, cioè il processo esaurisce il suo codice e quindi finisce (*exit*);
- termina il suo tempo di CPU, cioè il suo quanto di tempo, e quindi ritorna nella lista dei processi pronti RL (*ready list*);
- per poter evolvere necessita di una risorsa che al momento non è disponibile: il processo si sospende (*suspend*) e passa nello stato di attesa, insieme ad altri processi, formando la waiting list WL.

Quindi durante l'evoluzione il processo può trovarsi in uno dei seguenti tre stati:

- in esecuzione (running);
- pronto (ready);
- sospeso (waiting), anche detto bloccato (**blocked**), che è lo stato in cui si trova quando non può ottenere la CPU anche se questa è libera, poiché è in attesa di qualche evento esterno o risorsa che al momento non è disponibile.

> Dallo stato di sospeso, cioè dallo stato di attesa, un processo non può passare in quello di esecuzione: infatti, quando si rende disponibile la risorsa che sta "aspettando", viene spostato dalla WL ma viene inserito nelle RL, cioè nella lista dei processi pronti ad accedere alla CPU. Quando arriverà il suo turno, gli verrà assegnato il processore e solo allora potrà evolvere.

Sospensione per interrupt

Il SO ha un diverso comportamento nel caso che il processo venga sospeso a causa di una interruzione associata a un dispositivo I/O rispetto alla sospensione dovuta allo scadere del time slice: gli interrupt dovuti ai dispositivi di I/O sono organizzati in classi e a essi viene associata una locazione, spesso vicina alla parte bassa della memoria, chiamata interrupt vector, che contiene l'indirizzo della procedura di gestione delle interruzioni.

Se si verifica un'interruzione (causata per esempio da problemi sul disco fisso) quanto è in esecuzione il processo XXX, il *program counter* di questo processo, la *parola di stato* del programma e la *maschera delle interruzioni* (a volte anche uno o più registri) vengono messi sullo stack corrente dall'hardware dedicato alle interruzioni e la CPU salta all'indirizzo specificato nell'interrupt vector che provvede a completare il salvataggio dello stato del processo prima di eseguire la ruotine della interruzione.

Quando termina la gestione dell'interrupt, viene richiamato lo scheduler per vedere quale processo deve essere mandato in esecuzione e un piccolo programma assembler esegue il caricamento dei registri e la mappa di memoria per il nuovo processo corrente.

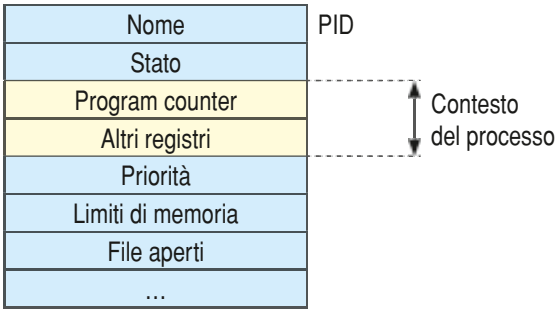
CONTEXT SWITCHING

Tutte queste azioni di salvataggio e ripristino vengono chiamate context switching.

PCB (Process Control Block)

Concludiamo questa sintesi sui processi ricordando la struttura del descrittore del processo PCB (Process Control Block):

- identificatore unico (PID);
- stato corrente;
- program counter;
- registri;
- priorità;
- puntatori alla memoria del processo;
- puntatori alle risorse allocate al processo.



Sappiamo che il program counter e i registri formano il contesto del processo: questi campi prendono anche il nome di area di salvataggio dello stato della CPU.

Oltre a queste informazioni sono anche presenti dati che riguardano *informazioni per l'accounting e per lo stato dell'I/O*, che riportano la lista dei file e delle periferiche associati al processo.
Il descrittore di processo viene allocato dinamicamente all'atto della creazione e opportunamente inizializzato. Viene rimosso dopo le operazioni di terminazione del processo.

Per implementare il modello a processi, il SO mantiene una tabella chiamata Process Table, contenente tutti i PCB dei singoli processi, in modo da avere sempre a disposizione le informazioni sullo stato del processo, aggiornandola quando il processo passa da uno stato di esecuzione a uno stato di pronto o bloccato, in maniera che possa essere fatto ripartire più tardi come se non fosse mai stato fermato.

Zoom su...

COMANDI PER LA CREAZIONE, SOSPENSIONE E TERMINAZIONE DEI PROCESSI

Nei sistemi ◀ *NIX Like ▶ la creazione di un processo avviene mediante la funzione:

```
pid fork();
```

All'atto della chiamata viene generato un nuovo PID per il figlio e un nuovo descrittore del processo PCB, vengono copiati nella

◀ *NIX Like Con *NIX si intendono i sistemi operativi UNIX e XENIX che hanno parecchie analogie e con *NIX Like i sistemi operativi che sono loro "somiglianti", come Linux che sappiamo essere direttamente derivato da UNIX. ▶

memoria del nuovo processo i segmenti dati e dati di sistema in modo da avere due coppie di segmenti identiche, dato che il "figlio" è un clone del "padre": l'unica differenza è il valore restituito dalla fork() stessa.

Il codice del processo padre viene condiviso dal figlio dal punto in cui viene invocata la fork() e quindi a partire da questa istruzione i due processi evolvono in parallelo eseguendo lo stesso codice: l'istruzione di fork() ritornerà il valore a entrambe le istanze del programma:

1. la prima istanza è il processo padre, con il valore del PID che viene assegnato dal SO al figlio;
2. la seconda istanza è il processo figlio appena creato, con valore pari a 0.

La terminazione di un processo avviene mediante la funzione:

```
void exit(int status);
```

Alla sua chiamata vengono eseguite le operazioni di chiusura dei file aperti, viene rilasciata la memoria, viene salvato il valore dell'exit status nel descrittore del processo in modo che potrà essere recuperato dal padre mediante le funzione *wait()* (di seguito descritta).

È importante osservare come il PID non viene rilasciato e il descrittore non viene distrutto, ma viene segnalata al processo padre la terminazione di un figlio.

Un processo padre può sospendere la propria attività in attesa che il figlio termini con l'istruzione:

```
pid wait(int* status);
```

Alla chiamata di questa funzione il processo padre si sospende in attesa della terminazione di un processo figlio: quando questo avviene, recupera il valore dello stato di uscita specificato dalla funzione *exit*() nel figlio.
In particolare il valore che viene restituito è composto da due byte:

- nel byte meno significativo della variabile viene indicata la ragione della terminazione, che può essere stata naturale o mediante un segnale;
- nel byte più significativo della variabile viene scritto il valore dello stato di uscita specificato nella funzione exit() del figlio

Il SO rilascia il PID del figlio, rimuove il suo descrittore di processo e restituisce il PID del figlio terminato.

Esiste anche la possibilità che un processo figlio termini prima che il padre abbia invocato la funzione *wait(),* il processo figlio diventa "defunct" o "zombie": in questo caso il processo è terminato ma il descrittore non può essere rilasciato.

Verifichiamo le conoscenze

Esercizi a scelta multipla

1 **Quale di queste affermazioni è errata?**

a) Il programma è un'entità passiva
b) Il programma è un'entità attiva
c) Il processo è un'entità passiva
d) Il processo è un'entità attiva

2 **In riferimento alla multiprogrammazione, quale delle seguenti affermazioni è errata?**

a) richiede la contemporanea presenza di più programmi in memoria
b) permette la contemporanea esecuzione di più processi
c) permette la contemporanea esecuzione di più istanze di un programma
d) permette la contemporanea esecuzione di più istanze di un processo

3 **Abbiamo quindi tre modelli di computazione per i processi (indica quelle errate):**

a) modello di computazione indipendente
b) modello di computazione dipendente
c) modello di computazione con cooperazione
d) modello di computazione con competizione
e) modello di computazione con integrazione

4 **In quale situazione può trovarsi un processo rispetto al processore (indica quella inesatta)?**

a) nuovo (new)
b) esecuzione (running)
c) attesa (sleeping)
d) pronto (ready-to-run)
e) finito (terminated)

5 **Nella multiprogrammazione più processi possono essere in running contemporaneamente:**

a) solo nei sistemi multiprocessori
b) solo nei sistemi con schedulazione
c) si, sempre
d) no, mai

6 **Dallo stato di esecuzione un processo può uscire per tre motivi (indica quello inesatto):**

a) termina la sua esecuzione;
b) produce un risultato errato (tipo divisione per 0);
c) termina il suo quanto di tempo;
d) gli manca una risorsa per evolvere

7 **In caso di interrupt da periferica, l'hardware effettua il salvataggio di:**

a) il program counter
b) i dati del processo
c) la parola di stato del programma
d) la maschera delle interruzioni
e) lo stato del processo

8 **Cosa significa l'acronimo PCB?**

a) Program Control Block
b) Process Control Block
c) Process CPU Block
d) Program CPU Block

Test vero/falso

1 Con throughput di sistema si intende il numero di programmi eseguiti per unità di tempo. V F
2 Un processo è un'entità logica in evoluzione. V F
3 Lo scheduling permette a più programmi di evolvere contemporaneamente. V F
4 Un processo è nello stato di pronto se ha tutte le risorse necessarie alla sua evoluzione. V F
5 Un processo è nello stato di attesa quando gli mancano almeno due risorse per poter evolvere. V F
6 Al nuovo processo viene assegnato un identificatore (PID, Process IDentifier). V F
7 Dallo stato di sospeso un processo può passare in quello di esecuzione se si rende disponibile la risorsa che sta "aspettando". V F
8 L'interrupt vector contiene l'indirizzo delle interruzioni. V F
9 La tabella chiamata Process Table contiene tutti i PCB dei singoli processi. V F
10 In *nix, la fork ritorna al processo figlio appena creato il valore PID=0. V F
11 Un zombie è un processo figlio senza il padre. V F

Esercizi a completamento

1 Disegna il diagramma degli stati di un processo descrivendone le singole transazioni.

2 Disegna la struttura del descrittore del processo PCB.

Domande a risposta aperta

1 Dai la definizione di processo.

2 Dai la definizione di multiprogrammazione.

3 Dai la definizione di processi cooperanti.

4 Dai la definizione di processi in competizione.

5 Dai la definizione di stato di un processo.

6 Dai la definizione di context switching.

7 Descrivi le seguenti istruzioni:

```
pid fork();
```

```
void exit(int status);
```

```
pid wait(int* status);
```

LEZIONE 2

RISORSE E CONDIVISIONE

IN QUESTA LEZIONE IMPAREREMO...

- il concetto di risorsa condivisa
- le richieste e le modalità di accesso alle risorse
- Il grafo di Holt per descrivere processi e risorse

Generalità

I processi sono i programmi in evoluzione e per poter evolvere hanno bisogno delle risorse del sistema di elaborazione: possiamo proprio vedere il sistema di elaborazione come composto da un insieme di risorse da assegnare ai processi affinché questi possano svolgere il proprio compito.

Una prima definizione di risorsa è la seguente:

RISORSA

Ogni componente riusabile o meno, sia hardware, sia software necessario al processo o al sistema.

Per accedere alle risorse i processi sono in competizione in quanto spesso esse non sono sufficienti per tutti e quindi è necessario "accaparrarsele" per poterle utilizzare; per esempio i processi competono per avere a disposizione la memoria RAM, per utilizzare l'interfaccia di rete, le stampanti ecc., e soprattutto il processore che, nelle nostre architetture, è singolo e senza di esso nessun processo può evolvere.

Le risorse possono essere suddivise in classi e le risorse appartenenti alla stessa classe sono equivalenti, per esempio bytes della memoria, stampanti dello stesso tipo ecc.

CLASSE E ISTANZE

Le risorse di una classe vengono dette istanze della classe; il numero di risorse in una classe viene detto molteplicità del tipo di risorsa.

Quando un processo necessità di una risorsa generalmente non può richiedere *una particolare risorsa* ma solo una istanza di una specifica classe: quindi una richiesta di risorse viene fatta per una classe di risorse e può essere soddisfatta da parte del SO assegnando al richiedente una qualsiasi istanza di quel tipo.

In altre parole, la molteplicità di una risorsa ci indica il numero massimo di processi che la possono usare contemporaneamente: se il numero di processi è maggiore della molteplicità di una risorsa, questa deve essere condivisa tra i processi che vi accedono concorrentemente.

ESEMPIO 1

Abbiamo detto che in un PC è presente un solo processore, quindi la molteplicità della risorsa processore è uguale a uno: il numero massimo di processi che possono *evolvere contemporaneamente* è quindi uno e quando abbiamo la necessità di far evolvere più processi assieme, questi condividono l'unica istanza della risorsa e competono per ottenerla.

Condivisione e gestione

Cerchiamo di chiarire meglio il concetto di condivisione prendendo spunto da una "legge ferroviaria" del secolo scorso:

◀ "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone" *Legge del Kansas* ▶

In questo caso i due treni *condividono l'incrocio*.

Gli esempi nella vita quotidiana di condivisione sono molteplici a partire da quelli di "natura stradale o ferroviaria" (incroci, posteggi, ponti ecc.) a quelli di natura sociale ("lo stesso bagno", "lo stesso tetto", "la stessa bandiera", "la stessa causa", la stessa opinione" ecc.).

Possiamo dire che condividere è sinonimo di "*avere in comune*" e quando parliamo di risorse di elaborazione intendiamo dispositivi hardware o componenti software che devono essere assegnati alternativamente ai singoli processi che le richiedono.

È quindi necessaria una gestione delle risorse che può essere organizzata in fasi, alcune delle quali sono di natura statica e riguardano la loro assegnazione (pianificazione), mentre altre sono di natura dinamica e sono relative al loro utilizzo (controllo):

- pianificazione della organizzazione:
  - allocazione;
  - disponibilità;
  - costo;
- controllo delle risorse:
  - controllo di accesso (locale o remoto);
  - ottimizzazione;
  - autenticazione;
  - controllo di correttezza operazioni ed eccezioni.

Le attività sopra elencate vengono svolte dal sistema operativo e per le risorse di molteplicità finita è necessario controllare gli accessi a ciascuna di esse in modo che il loro l'utilizzo risulti costruttivo.

Per ogni risorsa il SO mette a disposizione:
- un gestore della risorsa, che è un programma che ne regola il suo utilizzo;
- un protocollo di accesso alla risorsa, che consiste nella procedura con la quale un processo effettua la *richiesta* della risorsa, la *ottiene* e quindi la *utilizza* e alla fine la *rilascia* affinché gli altri processi la possano utilizzare.

Classificazioni

Tra processi e risorse esiste un legame molto stretto:

> I processi competono nell'accesso alle risorse, effettuando delle richieste per ottenere l'assegnazione di quanto gli necessita per poter evolvere.

In merito alla interazione tra risorse e processi possiamo effettuare la classificazione in base:
- al tipo di richieste;
- alla modalità di assegnazioni;
- alla tipologia delle risorse.

Classificazione delle richieste

Le richieste possono essere classificate secondo vari criteri.

A secondo il numero:
1. singola: la richiesta singola è il caso normale e si riferisce a una singola risorsa di una classe definita, cioè un processo richiede una sola risorsa alla volta.
2. multipla: si riferisce a una o più classi, e per ogni classe, a una o più risorse e deve essere soddisfatta integralmente; è il caso in cui un processo richieda contemporaneamente almeno due risorse per poter evolvere.

B secondo il tipo di richiesta che effettuano:
1. richiesta bloccante: è il caso in cui il processo necessità immediatamente di quella risorsa e se non gli viene assegnata immediatamente (in quanto occupata e quindi già in situazione di utilizzo da parte di atri processi) si sospende, passa nello stato di attesa e la sua richiesta viene accodata e riconsiderata dal gestore di quella risorsa ogni volta che viene rilasciata dal processo che la sta utilizzando.
2. richiesta non bloccante: in questo caso il processo può evolvere ugualmente e nel caso di mancanza di disponibilità gli viene effettuata una notifica che il processo richiedente esamina ma continuando la sua evoluzione senza cioè sospendere la propria elaborazione.

Classificazione dell'assegnazione

L'assegnazione delle risorse avviene in due modalità:

1. statica: l'assegnazione statica di una risorsa a un processo avviene al momento della creazione del processo stesso e rimane a esso "dedicata" fino alla sua terminazione; l'esempio più significativo è il descrittore di processo oppure l'area di memoria RAM nella quale è caricato (se non viene effettuato lo swapping);
2. dinamica: è il caso più frequente e generale nella naturale vita di un processo che avviene soprattutto per le risorse condivise che i processi richiedono durante la loro esistenza, le utilizzano quando sono a loro assegnate e le rilasciano quando non sono più necessarie oppure alla loro terminazione (esempio tipico sono le periferiche di I/O).

Classificazione delle risorse

Anche le risorse possono sottostare a varie classificazioni e tra esse ricordiamo le più importanti:

A in base alla mutua esclusività

1 risorse seriali: è il caso di risorse che non possono essere assegnate a più processi contemporaneamente ma questi devo attendere il loro turno per poterle utilizzare (si devono mettere in coda, cioè in "serie", uno dietro all'altro); questo tipo di risorsa si dice che ha accesso mutuamente esclusivo da parte dei processi, in quanto quando ne entra in possesso un processo gli altri devono aspettare che questo la rilasci per poterla utilizzare. Esempi tipici di risorsa con accesso seriale sono le stampanti e i CD-ROM.

2 risorse non seriali**:** ammettono l'accesso contemporaneo di più processi e quindi possono considerarsi risorse di molteplicità infinita, come per esempio i file a sola lettura, che possono essere letti contemporaneamente da un numero qualsivoglia di processi.

B in base alla modalità di utilizzo

1 risorse prerilasciabili: si dice prelilasciabile o preemptable una risorsa che mentre viene utilizzata da un processo può essere "liberata", cioè può essere sottratta al processo prima che abbia terminato di utilizzarla, senza che questo fatto danneggi il lavoro che stava effettuando e, pertanto, nel momento che gli viene restituita, esso può riprendere il lavoro dal punto in cui è stato interrotto senza "subire danni".

> Il processo che subisce il prerilascio forzato (o anticipato) deve sospendersi; la risorsa prerilasciata sarà successivamente restituita a quel processo che riprenderà la sua evoluzione dal punto in cui l'aveva interrotta.

Affinché una risorsa sia prerilasciabile deve avere le seguenti caratteristiche:
- il suo stato non si modifica durante l'utilizzo;
- il suo stato può essere facilmente salvato e ripristinato.

Gli esempi più "classici" di risorsa preemptive sono il processore e le aree di memoria.

Possiamo quindi definire una risorsa preemptable o rilasciabile come

RISORSA PREEMPTIVE O RILASCIABILE

Una risorsa si dice prerilasciabile se il suo gestore può sottrarla a un processo prima che questo l'abbia effettivamente rilasciata.

2 risorse non prerilasciabili: una risorsa si dice non prerilasciabile o non-preemptive se una volta assegnata a un processo non è possibile "sottrargliela" senza che si provochi un danno al compito che esso sta eseguendo, con il pericolo di dover ripetere completamente la sua esecuzione. Esempi tipici di risorse non-preemptive sono la stampante e il masterizzatore: se interrompiamo il processo che le sta utilizzando, molto probabilmente viene danneggiato, se non del tutto compromesso, il lavoro in fase di svolgimento.

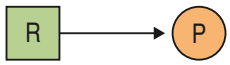
Grafo di Holt

Holt nel 1972 ha proposto un sistema di rappresentazione mediante un grafo che da lui ha preso il nome (grafo di Holt anche chiamato grafo di allocazione risorse o grafo delle attese) che permette di rappresentare tutte le situazioni in cui si possono venire a trovare i processi e le richieste di ri-

sorse, ed è particolarmente utile, come vedremo nelle prossime lezioni, per individuare situazioni di criticità tra processi e risorse.

Risorse e processi costituiscono due *sottoinsiemi* e sono rappresentati mediante nodi di due tipi:
- di forma quadrata le risorse (o di forma rettangolare nel caso di classi di risorsa e/o con risorsa multipla);
- di forma rotonda (cerchi) i processi.

Tra di essi vengono effettuati collegamenti orientati mediante archi:
- l'arco che connette una risorsa a processo indica che la risorsa è assegnata al processo



- l'arco che connette un processo a una risorsa indica che il processo ha richiesto la risorsa e che non gli viene assegnata dato che al momento della richiesta questa non è disponibile.

In questo modo si realizza un grafo orientato diretto (◀ **directed graph** ▶), con gli archi che hanno una sola direzione, e bipartito, in modo che non esistano archi che collegano nodi dello stesso sottoinsieme: gli archi possono solo connettere nodi di tipo diverso.

Se sono presenti più istanze della medesima classe di risorse, si effettua un partizione della risorsa stessa indicando la molteplicità con dei punti all'interno del box della risorsa (Grafo di Holt generale).

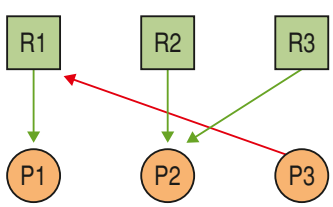
◀ **Directed graphs** ▶. The directed graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares. An arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process. Holt (1972) showed how the Coffman conditions that must hold for there to be a deadlock can be modeled using directed graphs. ▶

ESEMPIO 2

Nel primo esempio abbiamo tre processi (p1, p2 e p3) e tre risorse (R1, R2 ed R3) con molteplicità 1 che durante la loro evoluzione generano la seguente situazione:

```
P1 richiede R1            //gli viene assegnata
P2 richiede R2            //gli viene assegnata
P3 richiede R1            //NON gli viene assegnata, P3 rimane in attesa
```

La rappresentazione mediante il grafo di Holt è la seguente:



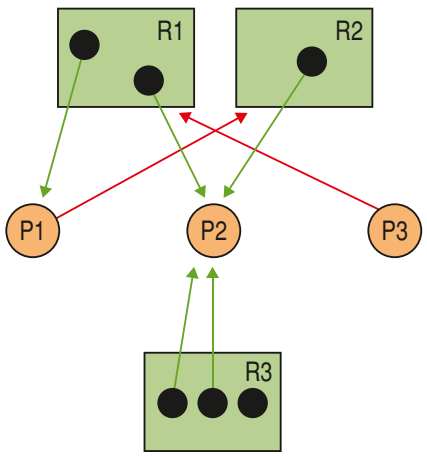
Supponiamo ora di avere classi di risorse con molteplicità diversa, per esempio:

- la classe R1: molteplicità 2
- la classe R2: molteplicità 1
- la classe R3: molteplicità 3

e la situazione è la seguente:

```
P1 richiede R1            //gli viene assegnata
P2 richiede R1            //gli viene assegnata
P2 richiede R2            //gli viene assegnata
P2 richiede R3            //gli viene assegnata
P2 richiede R3            //gli viene assegnata
P3 richiede R1            //NON gli viene assegnata, P3 rimane in attesa
P1 richiede R2            //NON gli viene assegnata, P1 rimane in attesa
```

La rappresentazione mediante il grafo di Holt è quindi:



Alcuni autori utilizzano una rappresentazione alternativa per indicare la molteplicità di risorsa utilizzata/richiesta da un processo indicando con un numero sulla freccia il grado di molteplicità e all'interno della classe il numero di risorse non ancora assegnate, cioè quante istanze di quella classe sono ancora disponibili.

Zoom su...

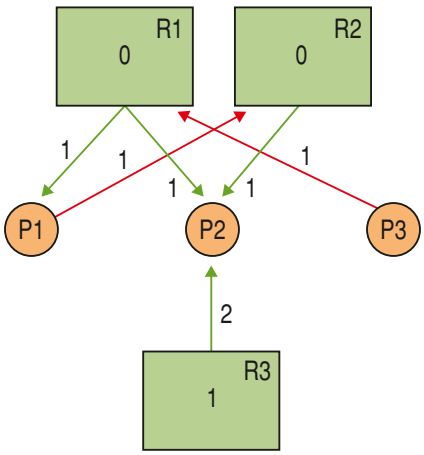
FORMALISMO DEI GRAFI DI HOLT

La scrittura formale dei due sottoinsiemi del grafo di Holt è la seguente:

- Set di vertici V = P U R dove:
  - P={P_0, P_1,…,P_n} (processi)
  - R={R_0, R_1,…,R_k} (classi di risorse)
- Set di archi **E** dove
  - arco di richiesta: $P_i \rightarrow R_j$ se un processo P_i ha richiesto un'istanza di risorsa R_j
  - arco di assegnazione: $R_j \rightarrow P_i$ se un'istanza di risorsa R_j è stata assegnata al processo P_i

ESEMPIO 3

L'esempio precedente sarebbe così rappresentato:



È importante sottolineare come nei grafi di Holt non si rappresentino le richieste che possono essere soddisfatte ma solo quelle pendenti: quindi le frecce uscenti dai processi verso le risorse indicano le "risorse mancanti" ai processi per evolvere, che sono in quel momento assegnate ad altri processi.

Riducibilità di un grafo di Holt

Spesso è utile avere una visione della situazione tra processi e risorse "spostata in avanti nel tempo", cioè trasformare il grafo di Holt in un grafo chiamato *ridotto* nel quale sono state tolte le situazioni in cui un processo è in grado di evolvere e che quindi sicuramente libererà le risorse che sta utilizzando a favore degli altri processi: siamo nel caso in cui un processo non attende nessuna risorsa e quindi graficamente *ha solo archi entranti* (risorse assegnate) e non ha archi uscenti (richieste in sospeso).

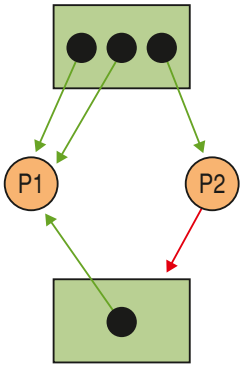
Il concetto fondamentale che sta alla base della riduzione di un grafo è la certezza che un processo che non ha bisogno di altre risorse per evolvere sicuramente prima o poi terminerà la sua elaborazione rilasciando le risorse che sta utilizzando e quindi non è un ostacolo per i processi che necessitano di quelle risorse e le stanno aspettando: sicuramente nel futuro prossimo le risorse saranno libere e il grafo ridotto evidenzia già questa situazione che, come vedremo in seguito, sarà utile perché la maggiore applicazione dei grafi di Holt è quella che ci permette di individuare situazioni critiche di blocco del sistema (stallo).

GRAFO RIDUCIBILE

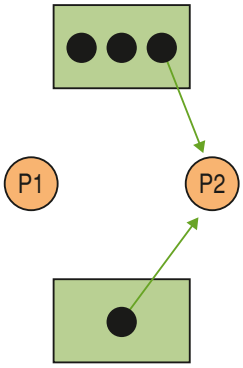
Un grafo di Holt si dice *riducibile* se esiste almeno un nodo di tipo processo con solo archi entranti.

ESEMPIO 4

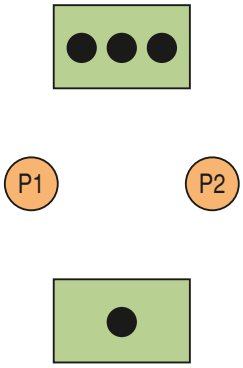
Nel seguente esempio abbiamo il processo P1 che sta utilizzando tre risorse e, dato che non è in attesa di altre, sta sicuramente evolvendo e, sicuramente, presto rilascerà quanto sta utilizzando:



effettuando la *riduzione per P1* si ottiene un nuovo grafo dove si considera terminata l'elaborazione di P1, si eliminano gli archi entranti e si rilasciano le risorse:



Ora anche il processo P2 può evolvere, dato che ha tutte le risorse che gli sono necessarie, e quindi possiamo anche effettuare la *riduzione per P2*, ottenendo:



Verifichiamo le conoscenze

Esercizi a scelta multipla

1 Quale tra le seguenti non è una fase di natura statica di gestione delle risorse?

a) allocazione b) disponibilità c) ottimizzazione d) costo

2 Quale tra le seguenti non è una fase di natura dinamica di gestione delle risorse?

a) allocazione
b) controllo di accesso
c) ottimizzazione
d) autenticazione
e) controllo di correttezza operazioni ed eccezioni

3 Le richieste di assegnazione delle risorse possono essere classificate:

a) secondo il numero
b) secondo il tempo di utilizzo
c) secondo il tipo di richiesta che effettuano
d) secondo il tempo di attesa

4 Le modalità di assegnazioni delle risorse possono essere classificate in:

a) statica
b) permanente
c) dinamica
d) condivisa
e) temporanea
f) esclusiva

5 Le risorse possono essere classificate:

a) in base al loro costo
b) in base alla mutua esclusività
c) in base alla modalità di utilizzo
d) in base alla loro molteplicità
e) in base alla loro velocità

6 Nei grafi di Holt (indica le affermazioni corrette):

a) risorse e processi costituiscono due sottoinsiemi e sono rappresentati medianti nodi
b) le risorse sono rappresentate mediante nodi di forma quadrata
c) le classi di risorsa sono rappresentate mediante nodi di forma quadrata
d) l'arco che connette una risorsa a processo indica che la risorsa è assegnata al processo
e) l'arco che connette un processo a una risorsa indica che il processo sta usando la risorsa

Test vero/falso

1 Il numero di risorse in una classe viene detto replicazione del tipo di risorsa. V F
2 Un processo all'occorrenza può richiedere una particolare risorsa di una classe. V F
3 Se il numero di processi è maggiore della molteplicità di una risorsa, questa viene condivisa. V F
4 I processi accedono concorrentemente alle risorse condivise. V F
5 Le fasi di pianificazione che riguardano l'assegnazione di una risorsa sono di natura dinamica. V F
6 Un processo effettua una richiesta multipla se necessita contemporaneamente di almeno due risorse per poter evolvere. V F
7 Una richiesta di risorsa è non bloccante se essa può essere utilizzata in modo condiviso. V F
8 Una periferica di I/O è un esempio tipico di risorsa assegnata staticamente. V F
9 Un esempio tipico di risorsa con accesso seriale è il CD-ROM. V F
10 Una risorsa si dice preemptable se può venire sottratta al processo che la sta utilizzando. V F
11 Nei grafi di Holt si rappresentano le richieste che possono essere soddisfatte. V F
12 Un grafo di Holt è *riducibile* se esiste almeno un nodo di tipo processo con solo archi entranti. V F

Verifichiamo le competenze

Disegna i grafi di Holt per le seguenti situazioni

1 Supponiamo di avere quattro processi A,B,C,D e quattro risorse R, S, T,Q. E supponiamo che il sistema allochi le risorse disponibili al primo processo che le richiede.

La sequenza di richieste è la seguente:

C richiede R
D richiede T
B richiede T
A richiede S
A richiede Q
A richiede R
C richiede T

Rappresenta la situazione mediante un grafo di Holt.

2 Supponiamo di avere tre processi A,B,C e tre risorse Q, R, S e che il sistema allochi le risorse disponibili al primo processo che le richiede.

La sequenza di richieste è la seguente:

1) A richiede R;
2) B richiede Q,
3) C richiede S;
4) C richiede R;
5) A richiede Q;
6) B rilascia Q;
7) B richiede S;
8) A rilascia R;

Rappresenta la situazione mediante un grafo di Holt.

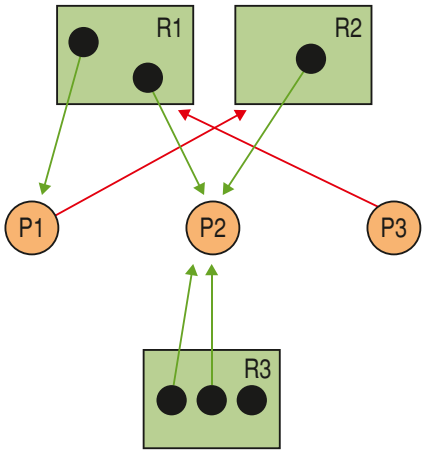
3 Supponiamo di avere tre processi A,B,C e tre risorse Q, R, S e che il sistema allochi le risorse disponibili al primo processo che le richiede.

La sequenza di richieste è la seguente:

1) A richiede Q;
2) C richiede S;
3) C richiede Q;
4) B richiede R;
5) B richiede S;
6) A rilascia Q;
7) A richiede S;
8) C richiede R.

Rappresenta la situazione finale mediante un grafo di Holt.

4 Dato il grafo di Holt di figura, per quale processo è possibile farne una riduzione? Disegna il grafo ridotto.



5 Supponiamo di avere un sistema dotato di 5 risorse (R,S,T,U,V) seriali, non prerilasciabili, contese da cinque processi (A,B,C,D,E). e che il sistema allochi le risorse disponibili al primo processo che le richiede.

La sequenza di richieste è la seguente:

B richiede R
A richiede R
A richiede S
C richiede T
A richiede T
E richiede U
D richiede V
C richiede U
E richiede R
D richiede T

1) Rappresentare lo stato del sistema con il grafo di Holt?

2) Per quale processo è possibile effettuare una riduzione? Rappresentare il grafo ridotto.

3) Sono possibili altre riduzioni? In caso affermativo procedere con tutte le successive riduzioni possibili.

LEZIONE 3

I THREAD O "PROCESSI LEGGERI"

IN QUESTA LEZIONE IMPAREREMO...

- la differenza tra processi e thread
- le modalità di utilizzo dei thread nei SO

Generalità

Le applicazioni che richiedono l'elaborazione parallela sono di tipologie molto diverse: dai videogiochi al controllo di processo, dall'elaborazione matematica alla gestione dei server Internet; in tutte queste situazioni la necessità di collaborare e di condividere risorse è decisamente diversa per ciascuna applicazione e quindi è opportuno avere a disposizione del progettista più strumenti per poter descrivere nella maniera più appropriata sia le situazioni in cui è richiesto un alto grado di parallelismo con molteplici risorse condivise, sia quelle in cui la cooperazione è molto ridotta e le attività svolte in parallelo sono quasi totalmente indipendenti, con poche interazioni e piccole aree di memoria condivise.

Per loro natura i processi sono entità autonome e sono quindi adatti alla descrizione di attività autonome e con poche risorse condivise e poco si prestano alla scrittura di applicazioni fortemente cooperanti.

Oltre al processo viene definita una nuova entità a esso molto simile ma con particolari caratteristiche che agevolano la risoluzione di problemi con alta cooperazione e condivisione di risorse: i thread.

Al modello a processi che implementa un insieme di macchine virtuali "una per ciascun processo", si affianca quindi un modello a thread, che definisce un sistema di macchine virtuali che realizzano "delle attività" piuttosto che un "compito completo", come descriveremo nel seguito.

In questa lezione riprenderemo i concetti principali sui processi e sull'utilizzo delle risorse in modo condiviso esposti nelle lezioni precedenti per confrontarli con i thread al fine di evidenziarne pregi e difetti per poter individuare le situazioni nelle quali è preferibile l'utilizzo di quest'ultimi rispetto ai processi.

Processi "pesanti" e "processi leggeri"

I processi che abbiamo descritto sino a ora vengono anche definiti processi pesanti per distinguerli dai thread che sono spesso chiamati processi leggeri.

Processi pesanti

Si è detto che il processo può essere visto come l'insieme della sua *immagine* e dalle *risorse* che sta utilizzando, dove:

- l'immagine del processo è costituita proprio dal *process ID*, *Program Counter*, *Stato dei Registri*, *Stack*, *Codice*, *Dati* ecc.;
- le risorse possedute sono i file aperti, processi figli, dispositivi di I/O..,

Abbiamo definito come spazio di indirizzamento l'insieme dell'immagine di un processo e le risorse da esso possedute: la sua allocazione dipende dalla tecnica di gestione della memoria adottata (per esempio, parte del codice può essere su disco e gestita con tecniche di paginazione e segmentazione o di overloading).

PROPRIETÀ DEI PROCESSI

Una caratteristica fondamentale dei processi è che ciascuno di essi ha un proprio spazio di indirizzamento, cioè due processi non condividono nessuna area di memoria: come vedremo in seguito, neppure i processi figli condividono le variabili dichiarate dei rispettivi padri che li hanno generati.

Quando un processo si sospende e avviene il cambio di contesto, per rendere operativo il nuovo processo le operazioni che il SO deve compiere sono molteplici e complesse in quanto richiedono il salvataggio del contesto del processo che si sospende e il ripristino di quello che inizia (o riprende) la sua esecuzione.

L'esecuzione delle operazioni di ◀ **context switch** ▶ richiede tempo utile di CPU che, quindi, viene così sprecato per effettuare queste operazioni "non produttive": questo tempo impiegato prende il nome di overhead e può essere definito come "costo di gestione" per realizzare il multitasking.

◀ **Context switch** Context switching is the procedure of storing the state of an active process for the CPU when it has to start executing a new one. ▶

Per questo motivo al processo viene "assegnato l'aggettivo di pesante" perché richiede "pesanti" elaborazioni per passare dallo stato di *pronto* a quello di *esecuzione*.

Naturalmente l'obiettivo di ogni SO è quello di ridurre al minimo l'overhead migliorando gli algoritmi di scheduling.

Processi leggeri

Il processo in evoluzione può essere visto come l'unione di due componenti:

- *le risorse che utilizza*, cioè quelle comprese nel suo spazio di indirizzamento;
- *l'esecuzione del codice*, cioè il flusso di evoluzione del programma che condivide la CPU con gli altri processi.

Il sistema operativo gestisce questi due componenti in modo indipendente, e questa osservazione è alla base dei thread:

- la *parte del processo* alla quale viene assegnata la CPU viene definita processo leggero (thread);
- la *parte del processo* che possiede le risorse viene definita processo pesante (processo o task)

THREAD

Un thread è un flusso di controllo che può essere attivato in parallelo ad altri thread nell'ambito di uno stesso processo e quindi nell'esecuzione dello stesso programma.

In altre parole, un thread è un "segmento di codice" (tipicamente una funzione) che viene eseguito in modo sequenziale all'interno di un processo pesante e tutti i thread definiti all'interno di un processo ne **condividono le risorse**, risiedono nello **stesso spazio di indirizzamento** e hanno accesso a **tutti i suoi dati**.

Ogni thread viene eseguito in parallelo agli altri thread mandati in esecuzione dallo stesso processo con il vantaggio, dato che sono tutti attivi nell'ambito dello stesso processo, di condividere lo spazio di indirizzamento e quindi le strutture dati.

Il termine "processo leggero" (LightWeight Process LWP)" vuole indicare che l'implementazione di un thread è meno onerosa di quella di un vero processo, e con multithreading si indica la molteplicità di flussi di esecuzione all'interno di un processo pesante.

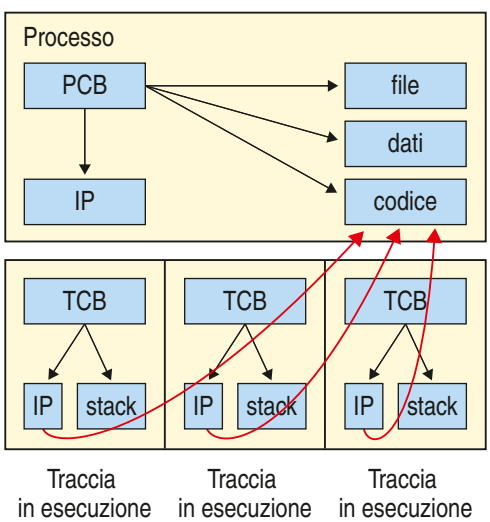
Per evolvere parallelamente agli altri thread o processi che si contendono la CPU, il thread mantiene comunque un insieme di caratteristiche comuni a processi pesanti, e in particolare:

- un identificatore di thread (ID);
- un program counter;
- un insieme di registri;
- uno stato di esecuzione (running, ready, blocked);
- un contesto che è salvato quando il thread non è in esecuzione;
- uno stack di esecuzione;
- uno spazio di memoria privato per le variabili locali.

I thread non hanno una loro area dove è presente il codice del programma in quanto condividono quello del processo che li genera così come ne condividono l'area dati.

Dati globali ed entità del Thread (Dati, Stack, Codice, Program Counter, Stato dei Registi) rappresentano lo stato di esecuzione del singolo thread.

Rappresentiamo graficamente tre thread mandati in esecuzione all'interno di un processo di cui eseguono tutti un segmento di codice:



Con TCB indichiamo il Thread Control Block, che è simile a PCB, e contiene i Registri, lo Stack, le variabili "locali" e lo stato esecuzione.

L'utilizzo dei thread offre la possibilità di sfruttare meglio le architetture multiprocessore e di comunicare e scambiare informazioni in modo immediato: inoltre è più vantaggioso avere i thread rispetto ai processi in quanto le operazioni context switch sono più semplici e veloci, dato che il thread condivide i vari dati, quindi sono in numero minore da memorizzare e ripristinare.

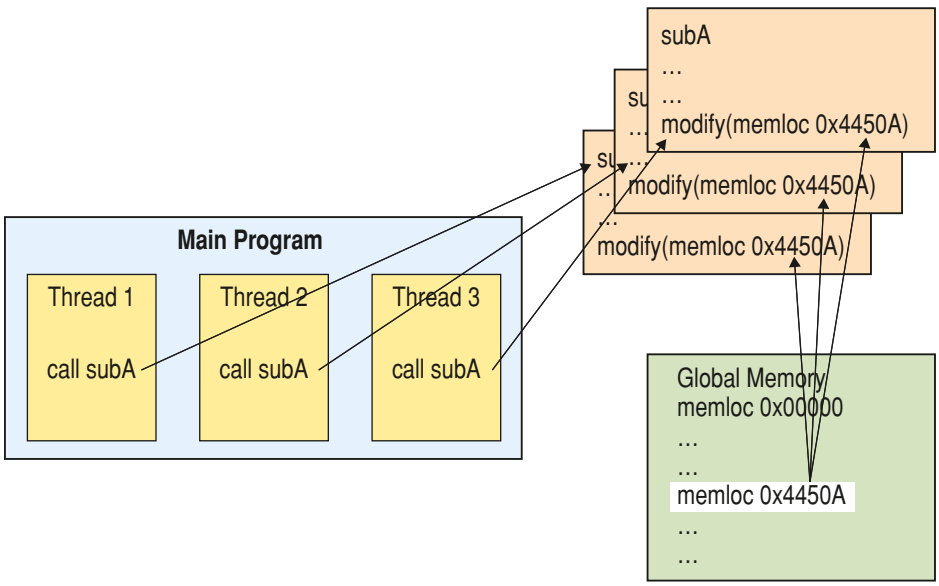
L'esecuzione dei thread richiede necessariamente che le routine di libreria debbano essere rientranti: i thread di un programma usano il SO mediante *system call* che usano dati e tabelle di sistema dedicate al processo e queste devono essere progettate in modo da poter essere utilizzate da più thread contemporaneamente (thread safe call) senza che vengano persi i dati.

TRHEAD SAFETY

Un programma o segmento di codice è detto thread-safe, o anche che gode della proprietà di thread safeness, se è corretto anche nel caso di esecuzioni multiple da parte di più threads garantendo che nessun thread possa accedere a dati in uno stato inconsistente, cioè durante il loro aggiornamento.

ESEMPIO 5

la funzione subA scrive il proprio risultato in una *variabile del processo* e restituisce al chiamante un puntatore a tale variabile.



Se due thread di uno stesso processo eseguono "nello stesso istante" la chiamata a due subA ognuno setta la variabile con un valore: quale valore sarà letto dai thread chiamanti al termine della esecuzione di subA?

La tabella seguente riporta il confronto tra processi e thread, evidenziando per ogni tipologia pregi e difetti.

	Processi	Thread
Creazione Distruzione	Richiedono allocazione, copia e deallocazione di grandi quantità di memoria	Richiedono solamente la creazione di uno stack per il thread
Errore	Non può danneggiare altri processi	Può danneggiare altri thread e l'intero processo cui appartiene
Codice	Un processo può modificare il proprio codice mediante il cambiamento di eseguibile	Il codice di un thread è fissato e presente nella sezione text del processo cui appartiene
Condivisione	È onerosa e deve essere implementata dal programmatore	È automaticamente garantita poiché tutti i thread condividono la memoria del processo cui appartengono
Mutua esclusione	La mutua esclusione è garantita automaticamente dall'isolamento proprio dei processi	Deve essere realizzata dal programmatore mediante semafori, mutex ecc.
Prestazioni	Limitate dall'overhead di gestione	Elevate
Concorrenza	Limitata dalla difficoltà di comunicazione	Elevate

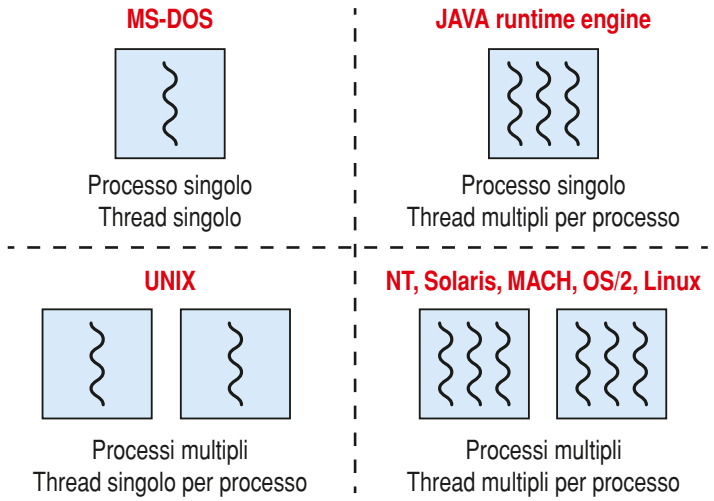
Soluzioni adottate: single threading vs multithreading

In base alla capacità di un sistema di gestire a livello kernel i thread, distinguiamo tra quattro possibili scenari, ottenuti dalla combinazione delle possibili situazioni:

- singolo processo e thread singolo;
- singolo processo e thread multiplo per processo;
- multiplo processo e thread singolo per processo;
- multiplo processo e thread multiplo per processo.

Queste quattro situazioni le possiamo riscontrare in quattro diversi ambienti operativi:

- MS-DOS: un solo processo utente e un solo thread;
- UNIX: più processi utente ciascuno con un solo thread;
- supporto run time di Java (JVM): un solo processo, più thread;
- Linux, Windows NT, Solaris: più processi utente ciascuno con più thread.



Con un normale personal computer è possibile avere a disposizione più ambienti operativi, sia partizionando il disco fisso e istallando un diverso sistema operativo su ogni partizione, sia istallando le macchine virtuali o gli emulatori: in questo volume faremo quindi riferimento sempre a situazioni con thread multipli, pensando a soluzioni con linguaggio Java a prescindere dal sistema operativo, dato che Java è per sua natura multipiattaforma, e a soluzioni realizzate in linguaggio C eseguite su macchine con sistema operativo Linux (o in emulazione).

Realizzazione di thread

Gli ambienti operativi hanno due modalità per realizzare un sistema multithreading a seconda che il kernel del sistema operativo sia o meno a conoscenza della loro esistenza, cioè se i thread vengono realizzati mediante chiamate al kernel oppure realizzati da software medianti procedure e librerie. Nel primo caso si parla di Kernel-Level, nel secondo di User-Level.

User-Level

I thread a livello utente sono quelli che vengono implementati grazie a delle librerie apposite (thread package) che contengono le funzioni per la creazione, terminazione, sincronizzazione dei thread e per realizzare anche i meccanismi per il loro scheduling: quindi nè la schedulazione e neppure lo switching con il cambio di contesto coinvolgono il nucleo, che "ignora" la loro presenza e gestisce solamente i processi.

I thread sono quindi di "proprietà e gestione" esclusiva del processo che li ha creati, uno alla volta, chiamando le apposite funzioni di libreria.

I vantaggi di questa soluzione sono innanzitutto l'efficienza di gestione in quanto i tempi di switching sono molto ridotti dato che non richiedono chiamate al kernel, hanno una grande flessibilità e scalabilità, dato che lo scheduling può essere modificato e dimensionato volta per volta in funzione della specifica situazione.
Dato che sono realizzati mediante librerie, possono essere implementati su qualunque sistema operativo e quindi hanno un alto grado di portatilità tra macchine e sistemi diversi.

Tra gli svantaggi il primo che riportiamo è il fatto che se un thread effettua una system call per esempio per motivi di I/O, oltre che a sospendere se stesso provoca la sospensione del processo che lo ha generato e quindi anche di tutti gli altri thread sempre generati dallo stesso processo.
Inoltre non è possibile sfruttare il parallelismo fisico in architetture multiprocessore per thread generati dallo stesso processo dato che sono "interni al processo stesso" e quindi assegnati a uno specifico processore.

L'esempio tipico di ambienti User-Level è la JVM (Java Virtual Machine) e quindi la possibilità di realizzare applicazioni multithread in linguaggio Java.

Kernel-Level

A livello di nucleo la gestione dei thread affidata al kernel tramite chiamate di sistema e quindi è il kernel che gestisce i thread come tutti gli altri processi, li deve schedulare, sospendere e risvegliare assegnandogli le risorse di sistema: a differenza del caso precedente, se un thread si sospende può naturalmente evolvere un secondo thread generato dallo stesso processo, in quanto sono tra loro schedulati in modo autonomo.

Questo è proprio il vantaggio più significativo di questa seconda soluzione unito al fatto che in architetture multiprocessori si può sfruttare al massimo il parallelismo fisico: inoltre lo stesso kernel può essere scritto come un sistema multithread.

Lo svantaggio principale è legato alla efficienza del sistema dovuta ai tempi impiegati dal kernel per il cambio di contesto durante la schedulazione che, di fatto, risulta essere più complessa in quanto deve gestire la copresenza di processi e thread.

Tra i sistemi operativi che realizzano ambienti Kernel-Level ricordiamo Linux, Unix e Windows.

Soluzione mista

Esistono anche soluzioni miste, come quella implementata in Solaris, che combina le proprietà di entrambi i meccanismi permettendo di creare a livello utente dei thread che solo però preventivamente devono essere definiti a livello di kernel (i thread utente vengono "mappati" sopra i thread a livello kernel, quindi non possono essere in numero superiore a essi), e lasciano all'utente le politiche di scheduling e di sincronizzazione.
I principali vantaggi sono che thread della stessa applicazione possono essere eseguiti in parallelo su processori diversi e che la chiamata al kernel da parte di un thread non blocca necessariamente il processo che lo ha generato.

Zoom su...

THREAD POSIX

Il modello ANSI/IEEE per i thread è definito dallo standard POSIX (Portable Operating System Interface for Computing Environments), che comprende un insieme di direttive per le interfacce applicative (API) dei sistemi operativi.

La definizione di standard ha lo scopo di indicare le regole da rispettare per realizzare strumenti compatibili, o meglio, dei programmi applicativi portatili in ambienti diversi: se un programma applicativo utilizza solamente i servizi previsti dalle API di POSIX può essere portato su tutti i sistemi operativi che implementano tali API.

I thread di POSIX sono chiamati Pthread, e saranno da noi utilizzati per la realizzazione dei programmi concorrenti semplicemente richiamando la libreria <pthread.h>.

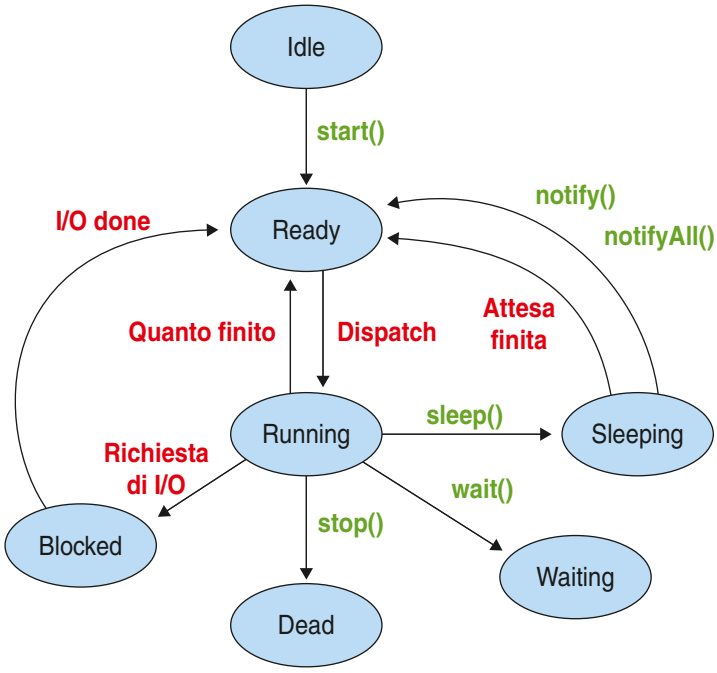
Stati di un thread

Come per i processi, anche i thread durante il loro ciclo di vita passano attraverso diversi stati: inoltre la vita del thread è legata alla vita del processo che li genera in quanto se un processo termina questo comporta anche la terminazione di tutti i suoi thread: è invece "indipendente" durante la vita anche se è attivo all'interno del processo, cioè evolve indipendentemente dal fatto che il processo sia in esecuzione o in attesa.

Il ciclo di vita dei thread è riportato nel diagramma della pagina seguente.

Nel diagramma *le transizioni sotto il controllo del sistema sono indicate in* rosso, quelle effettuate da istruzioni, e quindi dove il controllo è del programma, sono indicate in verde.

Gli stati sono i seguenti:
- Idle: prima di essere avviato;
- Dead: terminate le sue istruzioni;
- Blocked:. in attesa di completare l'I/O;
- Sleeping: sospeso un periodo;
- Waiting: in attesa di un evento;
- Running: in esecuzione;
- Ready: pronto per l'esecuzione ma in attesa della CPU.



Un thread è idle quando non è ancora avviato e viene posto nello stato di ready dove condivide una apposita coda con tutti i thread e i processi che attendono la CPU e sono gestiti dalle diverse politiche di scheduling, alternando Running a Ready.
Dall'esecuzione passa allo stato di Blocked in caso di richieste di operazioni di I/O, e nello stato di waiting quando esegue chiamate bloccanti che ne causano la sospensione: non appena la causa del blocco è stata rimossa (per esempio sono arrivati i dati dal disco) il thread ritorna nello stato di ready.
Dallo stato di running un thread può passare anche allo stato di sleeping, che è semplicemente uno stato nel quale effettua dei cicli di attesa (sospensione per un certo tempo) per poi essere riaccodato tra i processi pronti.
Dallo stato di running un thread può infine passare allo stato di dead, qualora esso abbia eseguito tutte le proprie istruzioni.

Quando un thread si blocca per una richiesta di I/O e passa nello stato di Blocked il sistema potrebbe decidere di eseguire un altro thread dello stesso processo oppure un thread di altri processi pronti, e la scelta del comportamento dipende dal tipo di implementazione
- nei Thread Java implementati a livello utente il sistema non vede gli altri thread di quel processo ed esegue un processo differente;
- nei Thread C implementati a livello kernel il sistema può gestire lo scheduling a livello thread secondo una qualche politica definita nel sistema operativo.

Utilizzo dei thread

Una delle principali applicazioni dei thread è quella di permettere di organizzare l'esecuzione di lavori con attività in foreground e in background: per esempio, mentre un thread gestisce l'I/O con l'utente, altri thread eseguono operazioni sequenziali di calcolo in background.

Per esempio nei fogli elettronici vengono utilizzati per le procedure di ricalcolo automatico, nei word processor per effettuare la reimpaginazione oppure il controllo ortografico del documento che si sta creando, nel web per effettuare le ricerche nei motori o nei database ecc.

Un altro importante utilizzo dei thread è quello di implementarli per l'esecuzione di attività asincrone, come per esempio le operazioni di garbage collection nella gestione della RAM oppure nelle procedure di salvataggio automatico dei dati (backup schedulati).

Verifichiamo le conoscenze

Esercizi a scelta multipla

1 Il processo può essere visto come l'insieme (due elementi):

a) della sua immagine
b) del suo stato
c) dalle risorse che sta utilizzando
d) del valore dei suoi registri

2 Quale tra le seguenti affermazioni relativa ai thread definiti all'interno di un processo è falsa?

a) ne condividono le risorse
b) ne condividono i registri
c) risiedono nello stesso spazio di indirizzamento
d) hanno accesso a tutti i suoi dati

3 Ogni thread ha un suo (indica la risposta errata):

a) identificatore di thread
b) program counter
c) insieme di registri
d) stato di esecuzione
e) segmento di codice
f) stack di esecuzione
g) spazio di memoria privato per le variabili locali

4 Ai seguenti ambienti operativi associa la capacità gestione a livello kernel:

1 ..... UNIX
2 ..... Supporto run time di Java (JVM)
3 ..... MS-DOS
4 ..... Linux, Windows NT, Solaris

a) singolo processo e thread singolo
b) singolo processo e thread multiplo per processo
c) multiplo processo e thread singolo per processo
d) multiplo processo e thread multiplo per processo

5 Gli ambienti operativi hanno due modalità per realizzare un sistema multithreading:

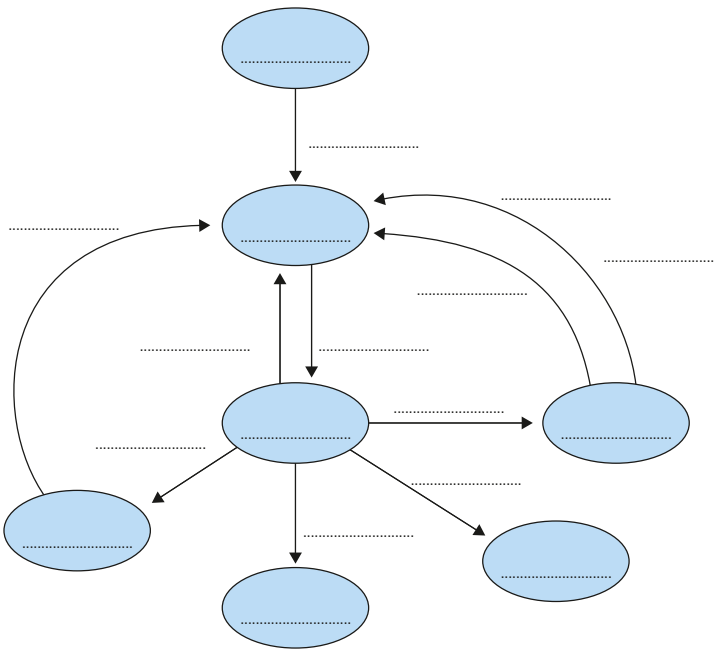
a) Kernel-Level
b) Thread-Level
c) System-Level
d) User-Level

Test vero/falso

1	I processi si prestano alla scrittura di applicazioni fortemente cooperanti.	V F
2	Due processi non condividono nessuna area di memoria.	V F
3	Due processi condividono le variabili dei loro antenati.	V F
4	Con overhead si intende il tempo sprecato per le operazioni di context switch nel multitasking.	V F
5	Ogni thread viene eseguito in parallelo agli altri mandati in esecuzione dallo stesso processo.	V F
6	Nei thread le operazioni context switch sono più semplici e veloci.	V F
7	L'esecuzione dei thread richiede che le routine di libreria debbano essere rientranti.	V F
8	Un programma ha la proprietà di thread safeness se è corretto anche nel caso di esecuzioni multiple da parte di più threads.	V F
9	I thread a livello utente sono anche chiamati User-Level perché vengono implementati grazie alle librerie del linguaggio utente.	V F
10	I thread Kernel-Level sono più efficienti dei thread User-Level.	V F
11	I thread Kernel-Level possono meglio sfruttare i multiprocessori rispetto ai User-Level.	V F
12	Nelle soluzioni miste possono esistere più threads che processi.	V F

Esercizi a completamento

1 Completa il seguente diagramma degli stati di un thread.



2 Descrivi i passaggi di stato.

Dall'esecuzione allo stato di Blocked.

Dall'esecuzione allo stato di waiting.

Dallo stato di running un thread può passare anche allo stato di sleeping.

Dallo stato di running un thread può passare allo stato di dead.

LEZIONE 4

ELABORAZIONE SEQUENZIALE E CONCORRENTE

IN QUESTA LEZIONE IMPAREREMO...

- il concetto di programmazione concorrente
- a realizzare il grafo delle precedenze
- il concetto di interazione tra processi

Generalità

La programmazione imperativa che si apprende nei primi corsi di informatica ha come riferimento un esecutore *sequenziale* che svolge una sola azione alla volta sulla base di un *programma sequenziale*.

ELABORAZIONE SEQUENZIALE

Con il termine elaborazione sequenziale si intende l'esecuzione di un programma sequenziale che genera un processo sequenziale con un ordinamento totale alle azioni che vengono eseguite.

Lo stesso teorema di Bhon e Jacopini indica nella sequenza una delle *tre figure strutturali fondamentali*.
L'elaborazione sequenziale è quindi un concetto fondamentale nell'informatica in quanto gli algoritmi che vengono computati sono composti da una sequenza finita di istruzioni in corrispondenza delle quali, durante la loro esecuzione, l'elaboratore passa attraverso una sequenza di stati (traccia dell'esecuzione del programma).

Anche l'esecutore dei programmi fino a ora utilizzato è una macchina sequenziale che si basa sul modello Von Neumann, cioè dotato di una sola unità di elaborazione (singola CPU).

Ma gli elaboratori non hanno tutti una sola CPU e inoltre, come abbiamo visto nello studio dei sistemi operativi, alcune attività del processore possono essere parallelizzate, come per esempio le lunghe fasi di input che provocano enorme spreco di tempo macchina soprattutto nei processi ad alta interattività con l'utente; esistono inoltre applicazioni che per loro natura necessitano di

attività parallele, come i server web, i video games a più giocatori, i robot, e per essi la codifica sequenziale delle attività propria della programmazione sequenziale non è in grado di descrivere con naturalezza queste situazioni.

Queste situazioni necessitano di un diverso modello in grado di effettuare la programmazione di un esecutore *concorrente*, ovvero di un elaboratore che è in grado di eseguire più istruzioni contemporaneamente.

PROGRAMMAZIONE CONCORRENTE

Con *programmazione concorrente* si indicano le tecniche e gli strumenti impiegati per descrivere il comportamento di più attività o processi che si intende far eseguire contemporaneamente in un sistema di calcolo (*processi paralleli*).

Siamo in una situazione di elaborazione contemporanea reale solo nel caso in cui l'esecutore sia dotato di una architettura multiprocessore, cioè con più processori che possono eseguire ciascuno un singolo programma: nei sistemi monoprocessori sappiamo che il parallelismo avviene solo virtualmente, grazie alla multiprogrammazione, e più processi evolvono "in parallelo" grazie al quanto di tempo che viene loro assegnato dalle politiche di scheduling del sistema operativo.

Il sistema operativo è per eccellenza l'esempio più eclatante di programmazione concorrente: il suo compito è quello di assegnare le risorse hardware dell'elaboratore ai processi utente che ne fanno richiesta, cercando di massimizzarne l'efficienza nella loro utilizzazione.
Le attività del sistema operativo devono essere eseguite concorrentemente in modo da consentire l'esecuzione contemporanea di più programmi utente: ogni attività interagisce con le altre sia in modo indiretto, occupando delle risorse comuni, sia in modo diretto, scambiando informazioni in merito allo stato delle risorse e dei programmi di utente al fine di realizzare la multiprogrammazione.

In un sistema multiprogrammato i programmi d'utente e le singole funzioni svolte dal sistema operativo possono essere considerati come un insieme di processi che competono per le stesse risorse.

Quindi un sistema multiprogrammato è un sistema concorrente, così definito:

SISTEMA CONCORRENTE

Per sistema concorrente intendiamo un sistema software implementato su vari tipi di hardware che "porta avanti" contemporaneamente una molteplicità di attività diverse, tra di loro correlate, che possono cooperare a un obiettivo comune oppure possono competere per utilizzare risorse condivise.

PROCESSO CONCORRENTE

Due processi si dicono concorrenti se la prima operazione di uno di essi ha inizio prima del completamento dell'ultima operazione dell'altro.

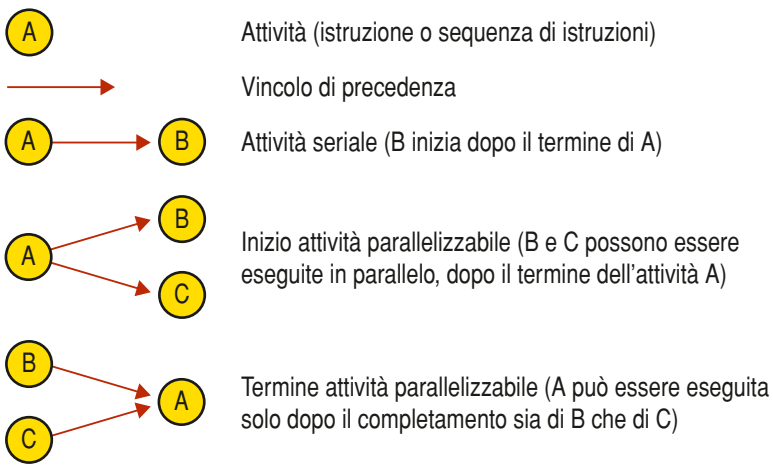
Processi non sequenziali e grafo di precedenza

Nei processi sequenziali la sequenza degli eventi che costituisce il processo è *totalmente ordinata*: se la rappresentiamo mediante un grafo orientato questo il grafo risulterà totalmente ordinato in quanto la sequenza degli eventi è ben determinata, cioè l'ordine con cui vengono eseguiti e sempre lo stesso.
Un grafo che descrive l'ordine con cui le azioni (o gli eventi) si eseguono del tempo prende il nome di grafo delle precedenze.

Nei processi paralleli, invece, l'ordinamento non è completo, in quanto l'esecutore per alcune istruzioni "è libero" di scegliere quali iniziare prima senza che il risultato sia compromesso: possiamo affermare che nella elaborazione parallela l'esecuzione delle istruzioni segue un ordinamento parziale.

Per descrivere questa libertà nella evoluzione dei processi concorrenti utilizziamo proprio il grafo delle precedenze (o diagramma delle precedenze): in un processo sequenziale il grafo delle precedenze degenera in una *lista ordinata* mentre in un processo parallelo è un *grafo orientato aciclico* e i percorsi alternativi indicano la possibilità di esecuzione contemporanea di più istruzioni.

Per la descrizione del grafo delle precedenze vengono utilizzati i seguenti simboli con i rispettivi significati:



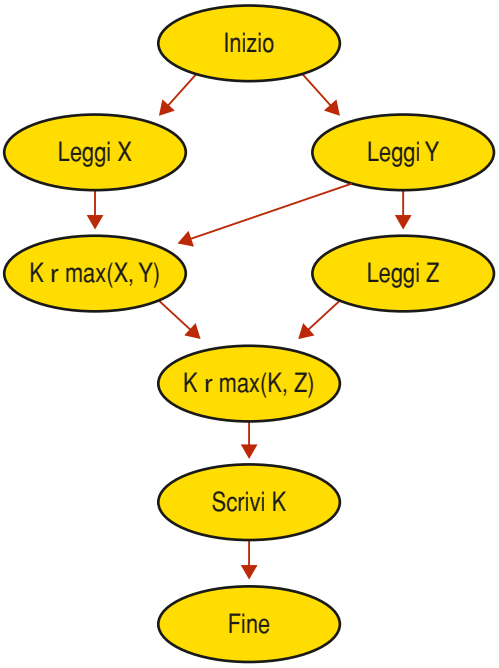
ESEMPIO 6 *Grafo delle precedenze*

Vediamo un semplice esempio di un algoritmo che legge tre numeri e ne individua il maggiore.
Per prima cosa scriviamo il codice sequenziale dell'algoritmo in *pseudocodifica (algoritmo sequenziale)*:

```
   inizio
1. leggi X;
2. leggi Y;
3. leggi Z;
4. K ← max(X;Y);
5. K ← max(K;Z);
6. scrivi K;
   fine
```

Ora riportiamo le istruzioni in un grafo dove esprimiamo i vincoli di effettiva precedenza per l'esecuzione delle istruzioni: *leggi X* e *leggi Y* non devono essere eseguite per forza in questo ordine, anzi, potrebbero anche essere effettuate in parallelo; anche la lettura di Z può essere eseguita dopo l'istruzione 4, oppure in parallelo con essa, mentre le istruzioni 5 e 6 devono per forza chiudere la sequenza delle operazioni.

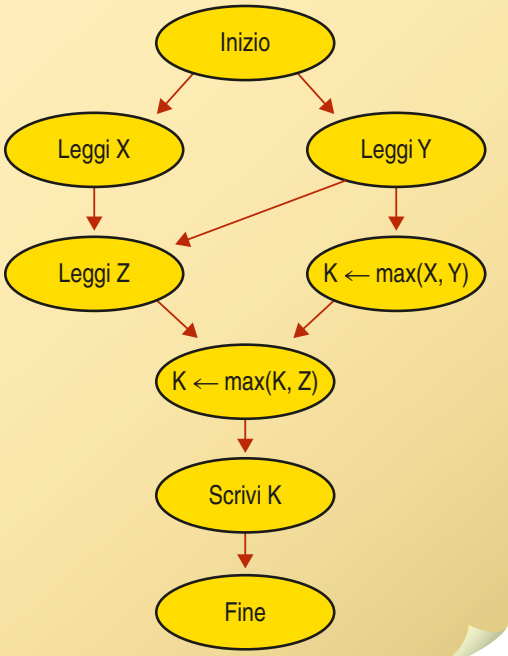
Il grafo è così fatto: ▶



Questo non è l'unico diagramma delle precedenze possibile: la lettura di Z potrebbe essere eseguita dal ramo che effettua la lettura di X e l'operazione di calcolo del massimo tra X e Y potrebbe essere eseguita al suo posto, ottenendo ▶

che è altrettanto logicamente corretto.

I grafi ottenuti sono quindi dei grafi a ordinamento parziale.



ESEMPIO 7

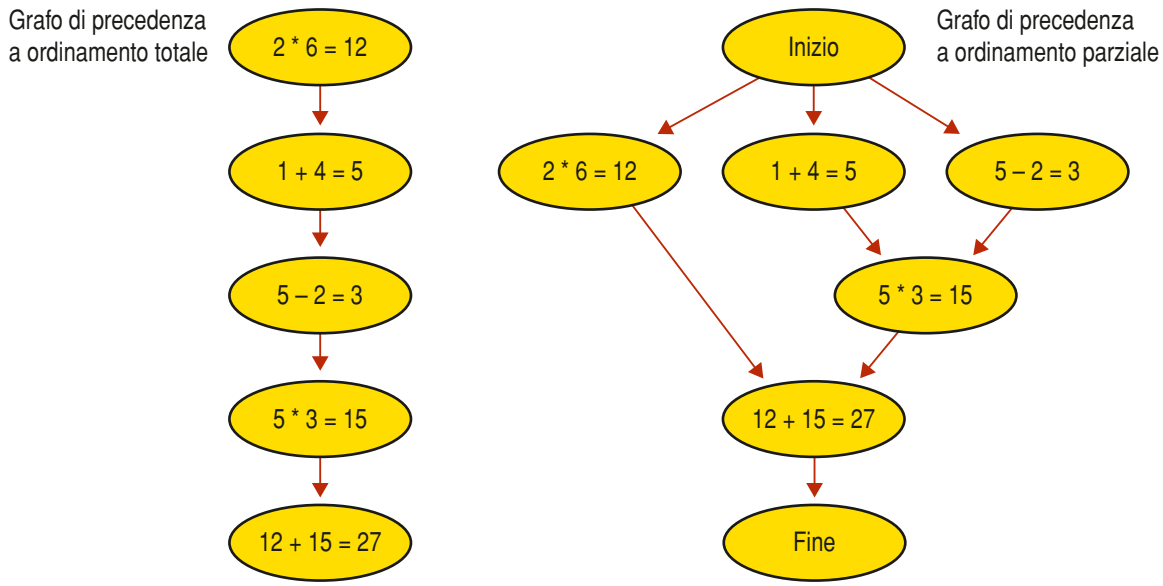
Ordinamento parziale e totale

Vediamo un secondo esempio, dove il problema da risolvere è quello di valutare una espressione matematica: (2 * 6) + (1 + 4) * (5 - 2)

Le regole di precedenza in questo caso sono dettate dalla matematica, dove dapprima si devono eseguire le operazioni all'interno delle parentesi e successivamente si deve rispettare le precedenza degli operatori (/, *, + e -).

Non esiste l'obbligo un ordinamento totale fra le operazioni da eseguire per il calcolo delle parentesi: potremmo indifferentemente eseguire prima (1 + 4) piuttosto che (2 * 6) o viceversa senza compromettere il risultato finale.

Rappresentiamo la soluzione con il grafo sequenziale a ordinamento totale per confrontarla con quello a ordinamento parziale, dove introduciamo la parallelizzazione delle attività (che in questo caso sono tutte operazioni algebriche):



Scomposizione di un processo non sequenziale

Un processo non sequenziale consiste nella elaborazione contemporanea di più processi che sono di tipo sequenziale, e quindi possono essere studiati, descritti e programmati singolarmente.

SCOMPOSIZIONE SEQUENZIALE

Un processo non sequenziale può essere scomposto in un insieme di processi sequenziali che possono essere eseguiti contemporaneamente.

Possiamo quindi affrontare lo studio dei processi paralleli scomponendoli in processi sequenziali e risolvendoli ciascuno separatamente, con la programmazione classica dei processi sequenziali.

Per poter correttamente descrivere la concorrenza è necessario distinguere le attività che i processi eseguono in due tipologie:

- attività completamente indipendenti;
- attività interagenti.

Processi indipendenti

La situazione più semplice da gestire è quella nella quale i processi sono tra loro completamente indipendenti.

PROCESSI INDIPENDENTI

L'evoluzione di un processo non influenza quella dell'altro.

Quindi sia che vengano eseguiti in sequenza che in parallelo non possono in nessun caso generare situazioni di funzionamento problematiche: l'unica accortezza è quella di rispettare per ogni processo l'ordine stabilito delle operazioni.

ESEMPIO 8 *Scomposizione in processi indipendenti*

Supponiamo di avere il seguente segmento di codice:

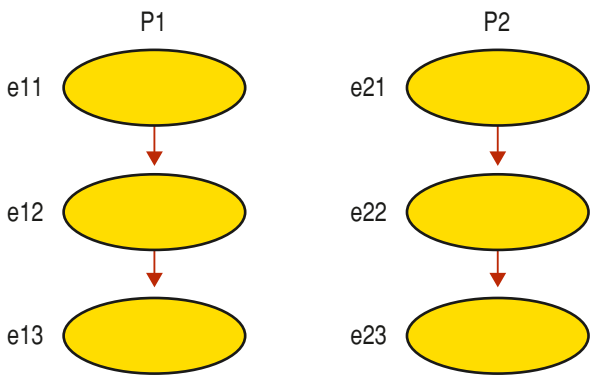
```
   inizio
1. leggi X;
2. leggi Y;
3. K ← SQRT(X);
4. W ← log(Y);
5. scrivi K;
6. scrivi W;
   fine
```

Possiamo scomporre il programma in due segmenti completamente indipendenti:

segmento 1	**segmento 2**
1. leggi X;	2. leggi Y;
3. K ← SQRT(X);	4. W ← log(Y);
5. scrivi K;	6. scrivi W;

Otteniamo due processi che eseguono ciascuno tre elaborazioni che *non hanno nulla in comune*, quindi ciascuno dei due processi può evolvere autonomamente senza interessarsi di quello che sta facendo l'altro. ▶

Il grafo di precedenze può essere scomposto in due grafi completamente autonomi.



Due processi indipendenti devono lavorare su un insieme privato di variabili e ogni variabile che ciascuno di essi modifica non può essere utilizzata da nessun altro processo: l'unico caso in cui due processi indipendenti utilizzano una variabile comune è quello in cui su tale variabile effettuano solo operazioni di lettura.

L'ultima osservazione che possiamo fare su un insieme di processi concorrenti disgiunti è che il risultato di ciascuno di essi è indipendente dalla velocità con la quale viene eseguito ma dipende unicamente dai dati di ingresso.

Processi interagenti

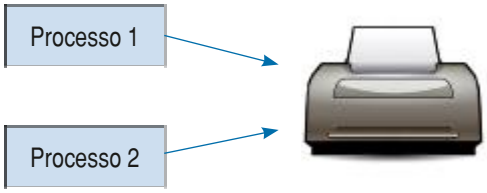
La seconda situazione è quella in cui i due (o più) processi non possono evolvere in modo completamente autonomo perché devono interagire o volontariamente o involontariamente e quindi la rappresentazione nel grafo delle precedenze dovrà necessariamente avere degli elementi comuni.

È possibile classificare le modalità di interazione tra processi in base alla loro conoscenza o meno della presenza degli altri.

A processi totalmente ignari
in questo caso i processi sono indipendenti e non sono stati progettati per lavorare assieme ma "evolvono" in un ambiente comune: interagiscono tra loro in competizione sulle risorse e si devono quindi sincronizzare.
Questa situazione viene gestita dal sistema operativo, che deve essere arbitro della loro evoluzione effettuando la sincronizzazione all'accesso delle risorse ogni volta che i processi le richiedono.

ESEMPIO 9 ***Processi in competizione***

I processi sono in competizione per l'accesso a una stampante



oppure per l'utilizzo di una tabella di dati o di file che non può essere letta da un processo mentre viene modificata da un altro.

B processi indirettamente a conoscenza uno dell'altro
è questa la situazione nella quale i processi sono a conoscenza dell'esistenza degli altri ma non ne conoscono il nome (o il ◀ **PID** ▶) e non possono comunicare direttamente tra loro ma devono cooperare per qualche motivo e possono scambiarsi i dati utilizzando risorse comuni, come aree di memoria condivisa.
Il sistema operativo deve offrire dei meccanismi di sincronizzazione che rendano possibile la cooperazione;

◀ **PID** In *nix, a PID is a process ID. It is generally a 15 bit number, and it identifies a process or a thread. ▶

C processi direttamente a conoscenza uno dell'altro
in questa situazione i processi devono cooperare per qualche motivo ma comunicano tra loro conoscendo i propri nomi: possono quindi effettuare direttamente lo scambio di informazioni mediante invio di messaggi espliciti.

> Il sistema operativo deve offrire dei meccanismi di comunicazione che rendano possibile la cooperazione.

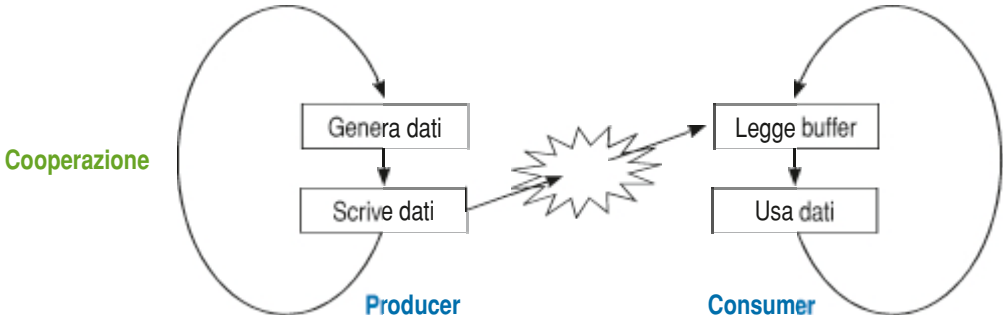
Meccanismi di comunicazione e sincronizzazione tra entità (anticipazioni)

Negli ultimi due casi i processi devono cooperare, quindi il sistema operativo deve offrire i meccanismi di sincronizzazione in modo da garantire il corretto funzionamento regolando e gestendo i vincoli sull'ordine con cui devono essere eseguite le operazioni.

> Una scorretta sincronizzazione dei processi dà luogo a una particolare categoria di errori, gli errori dipendenti dal tempo: questo tipo di errori spesso sono di difficile individuazione anche perché legati alla velocità relativa dei singoli processi e alla loro evoluzione autonoma, e quindi potrebbero manifestarsi o meno a seconda della casistica delle alternative di computazione in base alle diverse istanze di esecuzione.
> Un errore dipendente dal tempo potrebbe non ripetersi anche riavviando il sistema e riportandosi nelle medesime condizioni nelle quali si è manifestato una prima volta.

È quindi di fondamentale importanza la scelta di tecniche corrette di sincronizzazione che possono essere realizzate in tre modalità differenti:

A attraverso l'utilizzo di aree dati comuni (memoria condivisa, in inglese Shared Memory): un processo produce un dato (producer o produttore) e lo scrive nella memoria condivisa (buffer comune) in modo che l'altro processo (consumer o consumatore) lo possa leggere e utilizzare.



Spesso questa situazione viene regolata mediante un meccanismo chiamato monitor: questo si occupa di gestire la memoria in modo sincronizzato ricevendo dati creati dal *produttore* e gestendo le richieste di lettura e di risposta del *consumatore*.

B attraverso lo scambio di messaggi un processo trasmette le informazioni all'altro processo: il meccanismo a disposizione è realizzato con dei meccanismi simili a semplici operazioni di I/O che prendono il nome di InterProcess Communication (IPC).
La comunicazione diretta viene realizzata con due primitive dove ogni processo deve specificare il "nome" dell'altro processo con il quale deve comunicare

```
send(processo_destinatario, messaggio)
receive(processo_mittente, messaggio)
```

Questo è il meccanismo di più basso livello e qualsiasi sistema di interconnessione è in grado di supportarlo e può essere realizzato con due tipi di messaggi:

- messaggi sincroni: il mittente si ferma in attesa della risposta;
- messaggi asincroni: il mittente prosegue nella sua esecuzione e periodicamente (*polling*) o mediante un meccanismo di wake up (*eventi*) verifica le risposte ricevute.

C attraverso la chiamata a Procedura Remota (RPC Remote Procedure Call) un processo ha la possibilità di invocare una funzione di un servizio remoto come fosse una libreria locale: possiamo vedere questo meccanismo come una estensione del concetto client-server dove il server mette a disposizione le procedure remote che il processo cliente può invocare e tramite queste scambia informazioni o accede ad aree di memoria condivisa.

La bufferizzazione

Sia nella comunicazione diretta, sia in quella indiretta, i messaggi scambiati dai processi che comunicano risiedono in una coda temporanea.
Vi sono tre modi per implementare questa coda a seconda delle dimensioni del buffer di comunicazione:

1. *capacità zero*: in questo caso il mittente deve bloccarsi finché il destinatario riceve il messaggio;
2. *capacità limitata*: se il buffer è pieno il mittente deve bloccarsi;
3. *capacità illimitata*: il mittente non si blocca mai.

In base alla capacità del buffer il mittente non solo deve sincronizzarsi per accedere alla risorsa buffer ma a seconda del tipo di buffer ha dei comportamenti completamente differenti che vanno valutati caso per caso e che descriveremo nel seguito della nostra trattazione.

Verifichiamo le conoscenze

Esercizi a scelta multipla

1 Un'attività di un programma interagisce con le altre in modo diretto:

a) occupando delle risorse comuni
b) scambiando informazioni

2 Un'attività di un programma interagisce con le altre in modo indiretto:

a) occupando delle risorse comuni
b) scambiando informazioni

3 In un sistema concorrente le diverse attività tra di loro correlate (indica l'affermazione errata):

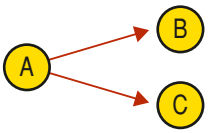
a) possono cooperare
b) possono competere
c) possono ostacolarsi
d) nessuna delle precedenti

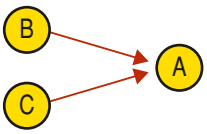
4 Il grafo delle precedenze:

a) può essere una lista ordinata
b) può essere una lista non circolare
c) può essere grafo ciclico
d) può essere grafo orientato aciclico

5 Abbina ai simboli il loro significato.

1 ..... 

2 ..... 

3 ..... 

a) inizio attività parallelizzabile

b) termine attività parallelizzabile

c) attività seriale

6 I grafi a ordinamento parziale:

a) sono incompleti
b) sono errati
c) descrivono un programma parallelo
d) descrivono espressioni con più soluzioni

Test vero/falso

1	Un esecutore sequenziale svolge una sola azione alla volta di programma sequenziale.	V F
2	Un processo sequenziale ha un ordinamento totale alle azioni che vengono eseguite.	V F
3	Un elaboratore in grado di eseguire più istruzioni contemporaneamente si dice concorrente.	V F
4	I processi paralleli vengono descritti con la programmazione concorrente.	V F
5	Per computare i processi paralleli sono necessarie macchine multiprocessore.	V F
6	Nei processi sequenziali la sequenza degli eventi è totalmente ordinata.	V F
7	Il grafo delle precedenze descrive l'ordine con cui le azioni si eseguono.	V F
8	Il grafo delle attività descrive l'ordine con cui le azioni si eseguono.	V F
9	Nei processi paralleli l'ordinamento delle istruzioni non è completo.	V F
10	Generalmente esiste un solo diagramma delle precedenze possibile.	V F

Verifichiamo le competenze

1 Costruisci un grafo delle precedenze che esprima il massimo grado di parallelismo nel calcolo delle seguenti espressioni dopo avere letto da input il valore di ogni variabile:

a) `(A+B)*(C+D)`
b) `(-B-SQRT(B2-4*A*C) )/(2*A)`
c) `2P=2A+2B S=A*B-B*C-C*D`
d) `(-B-SQRT(B2-4*A*C) )/(2*A)`

2 Costruisci il grafo delle precedenze dell'algoritmo che legge un numero e ne visualizza le prime 4 potenze utilizzando solo l'operatore di moltiplicazione.

Per esempio, l'algoritmo sequenziale è il seguente:

```
   inizio
1. leggi X;
2. scrivi X;
3. X2 ← X * X;
4. scrivi X2;
5. X3 ← X2 * X;
6. scrivi X3;
7. X4 ← X2 * X2;
8. scrivi X4;
   fine
```

3 Dato il seguente segmento di codice parallelizzabile:

A ← 10
B ← 20
G ← 2+B
D ← A+3
N ← A+3
E ← A+B
F ← D+N
H ← 2+N
L ← G+F+H
M ← 100+L

disegna il grafo delle precedenze che esprima il massimo grado di parallelismo.

4 Dato il seguente segmento di codice parallelizzabile:

A ← 10
B ← A+1
G ← 20+D
D ← A+3
N ← A+3
E ← B+D
F ← G+E+N
H ← 2+N
L ← G+F+H
M ← 100+L

disegna il grafo delle precedenze che esprima il massimo grado di parallelismo.

5 Dato il seguente segmento di codice parallelizzabile:

A ← A+3
A ← 2
B ← A+2
C ← B*3
D ← A+10
D ← D/12
E ← D+10
F ← A+10
H ← C+10+E+F
G ← 10+F
K ← W+H+G
L ← K+66
W ← 10+C

disegna il grafo delle precedenze che esprima il massimo grado di parallelismo.

6 Dato il seguente segmento di codice parallelizzabile:

A ← 2
B ← A+2
C ← A*3
D ← A+10
E ← D +B
E ← E+10
F ← B+C+10
G ← E+F

disegna il grafo delle precedenze che esprima il massimo grado di parallelismo.

7 Dato il seguente segmento di codice parallelizzabile:

A ← 22
B ← A+2
C ← B*3
E ← C+D
F ← A +2
D ← B+10
H ← P+1
H ← 2+M
N ← P+4
G ← H+N
F ← E+G

disegna il grafo delle precedenze che esprima il massimo grado di parallelismo.

LEZIONE 5

LA DESCRIZIONE DELLA CONCORRENZA

IN QUESTA LEZIONE IMPAREREMO...

- l'istruzione fork-join
- l'istruzione cobegin-coend

Esecuzione parallela

L'esecuzione di un processo non sequenziale richiede un sistema specifico che permetta la codifica e l'esecuzione dei programmi, cioè necessita di:

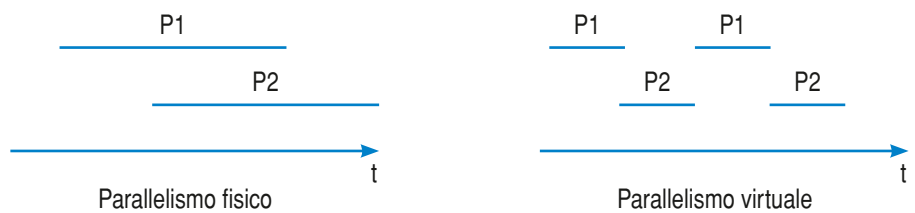
- un elaboratore non sequenziale;
- un linguaggio di programmazione non sequenziale.

Elaboratore non sequenziale

L'elaboratore non sequenziale è una macchina che deve essere in grado di eseguire più operazioni contemporaneamente e, come abbiamo detto, sostanzialmente questo si può ottenere in due modi differenti:

- architettura parallela, cioè sistemi multielaboratori;
- sistemi monoprocessori multiprogrammati.

Nei primi il parallelismo è fisico, mentre nei secondi il parallelismo è virtuale.



Nei nostri esempi tratteremo sempre il secondo caso in modo da poter scrivere e collaudare i programmi sui nostri PC che hanno tutti un sistema operativo multiprogrammato.

Linguaggi non sequenziali

Per scrivere programmi non sequenziali (o concorrenti) è inoltre necessario avere a disposizione dei particolari costrutti che permettano la descrizione delle attività parallele: non tutti i linguaggi hanno tali figure strutturali e quelli che consentono la descrizione delle attività concorrenti prendono il nome di linguaggi di programmazione concorrente (o non sequenziale).

La scrittura di un programma concorrente si basa sul concetto di scomposizione sequenziale.

Quindi il programmatore deve dapprima individuare le attività parallelizzabili e descriverle in termini di sequenze di istruzioni (blocchi sequenziali) e, successivamente mediante i linguaggi concorrenti, descrivere come tali moduli possono essere eseguiti in parallelo.

SCOMPOSIZIONE SEQUENZIALE

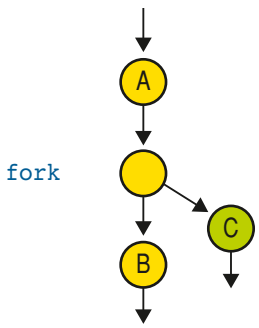
Un processo non sequenziale può essere scomposto in un insieme di processi sequenziali che possono essere eseguiti contemporaneamente.

I linguaggi di programmazione concorrenti di alto livello contemplano nuove istruzioni come il costrutto fork-join e cobegin-coend, che permettono di dichiarare, creare, attivare e terminare processi sequenziali.

Fork-join

Le istruzioni fork e join furono introdotte nel 1963 da Dennis e VanHorne per descrivere l'esecuzione parallela di segmenti di codice mediante la scomposizione di un processo in due processi e la successiva "riunione" in un unico processo.

Fork



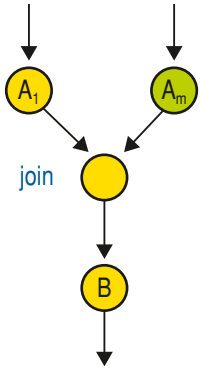
In rifermento al grafo delle precedenze, la fork corrisponde alla biforcazione di un nodo in due rami: l'esecuzione di una fork coincide con la creazione di un processo che inizia la propria esecuzione in parallelo con quella del processo chiamante.

Con un linguaggio di pseudocodifica è possibile tradurre il grafo nel seguente segmento di codice:

```
/* processo padre: */
inizio
…
A: <istruzioni>;
p2 = fork figlio1;
 B: <istruzioni>;
…
fine.

/* codice nuovo processo:*/
void figlio1()
{
C: <istruzioni>;
}
```

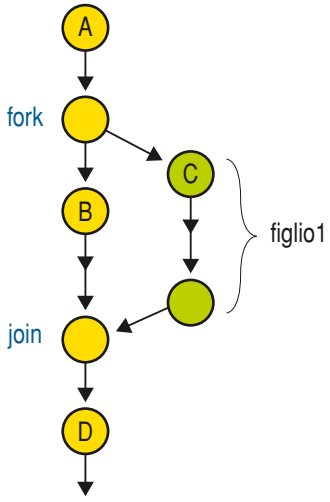
Join

La join è l'istruzione che viene eseguita quando il processo creato tramite la fork, ha terminato il suo compito, si sincronizza con il processo che lo ha generato e termina la sua esecuzione. ▶
Con un linguaggio di pseudocodifica è possibile tradurre il grafo nel seguente segmento di codice:




```
…
join p2;
B:<istruzioni>
…
```

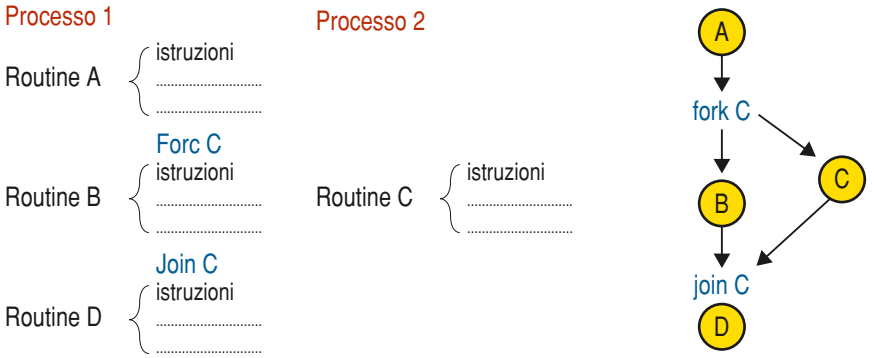
Il programma completo rappresentato nel diagramma a lato ▶ viene codificato in pseudolinguaggio con:




```
/* processo padre: */
inizio
…
A:<istruzioni>;
p2 = fork figlio1;      // inizia l'elaborazione parallela
   B: <istruzioni>;
join p2;                // termina l'elaborazione parallela
D:<istruzioni>;
…
fine

/* codice nuovo processo:*/
void figlio1 ()
{
   C:<istruzioni>;
}
```

Se ipotizziamo che ogni nodo sia una *routine*, possiamo meglio comprendere il funzionamento della istruzione fork con il seguente schema, dove sono messi in evidenza i due processi:

- P1 esegue la Routine A;
- con la fork attiva il processo 2 e in parallelo si eseguono la routine B e la routine C;
- P1 attende con la join la terminazione di P2;
- P1 esegue la routine C.



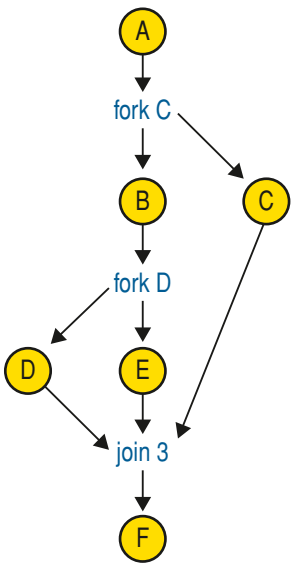
In questa definizione, la fork restituisce un identificatore di processo, che viene successivamente accettato da una join, in modo che sia specificato con quale processo si intende sincronizzarsi. Lo svantaggio di questa definizione è che la join può ricongiungere *solo due flussi di controllo*.

Join (count)

Esiste anche una formulazione estesa dell'istruzione join:

```
join(count);
```

dove count è una variabile intera non negativa e indica il numero di processi che si "riuniscono" in quel punto: viene utilizzata nel caso di terminazione congiunta di un numero superiore a due processi, come nel seguente grafo delle precedenze ▶



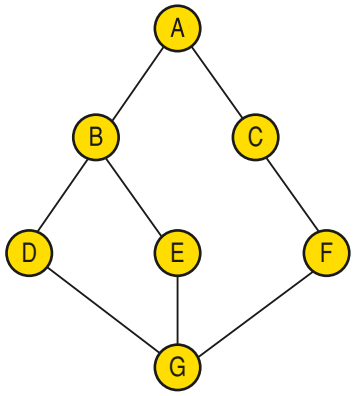
che viene codificato con questo segmento di programma:

```
/* processo padre: */
inizio
…
A:<istruzioni>;
p2 = fork C;            // inizia l'elaborazione parallela con P2
   B: <istruzioni>;
   p3 = fork D;         // inizia l'elaborazione parallela con P3
      E: <istruzioni>;
join 3;                 // termina l'elaborazione parallela di 3 processi
F:<istruzioni>;
…
fine.
```

Nella letteratura la sintassi della pseudocodifica a volte è leggermente differente da quella appena presentata: spesso viene semplificata omettendo le <istruzioni> e indicando semplicemente con le lettere maiuscole A,B,C oppure con S1,S2,…, Sn il blocco o la sequenza di istruzioni, e tale simbologia sarà adottata anche da noi per le prossime codifiche.

A volte viene anche messa una label per indicare dove il processo termina ed effettua la join, come nel seguente esempio: ▶

```
begin
   cont : = 3 ;
   A ;
   FORK E1 ;
   B ;
   FORK E2 ;
   D ;
   goto E3 ;
E1 : C ;
   F ;
   goto E3 ;
E2 : E ;
E3 : JOIN cont ;
   G ;
end.
```



Il linguaggio di shell UNIX/Linux ha due system call *simili* ai costrutti di alto livello definiti da Dennis e VanHorne:

- fork(): è l'istruzione utilizzata per creare un nuovo processo;
- wait(): corrisponde alla istruzione join e viene utilizzata per consentire a un processo di attendere la terminazione di un processo figlio.

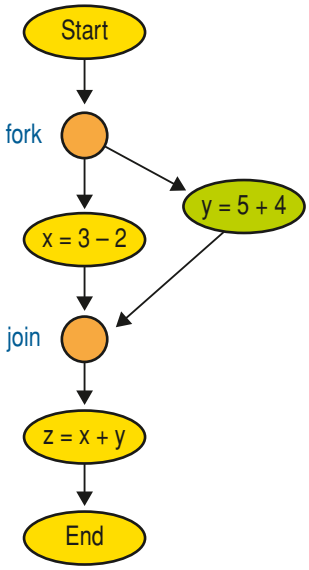
Sono però concettualmente diverse dato che sono *chiamate di sistema* mentre quelle da noi descritte sono istruzioni di un ipotetico *linguaggio concorrente di alto livello*: possono essere comunque utilizzate didatticamente per realizzare programmi concorrenti e alcuni esempi di codifiche in linguaggio C che le utilizzano in ambiente Linux sono riportati a titolo di esempio in una lezione dedicata al laboratorio a conclusione di questa unità di apprendimento.

ESEMPIO 10

Scriviamo un programma parallelo che esegue la seguente espressione matematica:

$$z = (3\text{-}2) * (5+4)$$

Le operazioni tra parentesi possono essere parallelizzate ottenendo il seguente grafo: ▶



che viene codificato in:

```
/* processo padre: */
inizio
   p2 = fork(figlio1);      // inizia l'elaborazione parallela
   x=3-2;
   join p2;                 // termina l'elaborazione parallela
   z=x+y;
…
fine

/* codice nuovo processo:*/
int figlio1()
{
   y=5+4;
   return y
}
```

Cobegin-coend

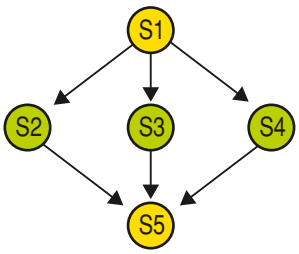
Il secondo costrutto che utilizziamo per descrivere la concorrenza è il cobegin-coend: è un costrutto che permette di indicare il punto in cui N processi iniziano contemporaneamente l'esecuzione (cobegin) e il punto che la terminano, confluendo nel processo principale (coend).

La sintassi del comando è:

```
cobegin <elenco delle attività parallele>
coend
```

ESEMPIO 11

Il grafo delle precedenze di figura: ▶
può essere descritto mediante il costrutto cobegin-coend con la seguente pseudocodifica:



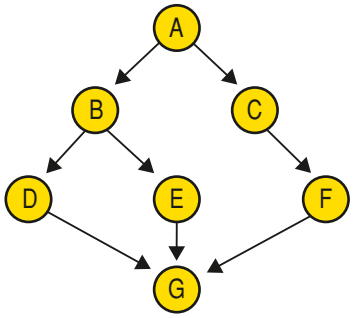

```
inizio
   S1
   cobegin
      S2        //attività parallele
      S3
      S4
   coend
   S5
fine
```

Al termine dell'esecuzione della sequenza S1 vengono attivati tre processi che eseguono in parallelo le sequenze di istruzioni S2, S3, S4: il processo padre S1 si sospende e rimane in attesa della loro terminazione.

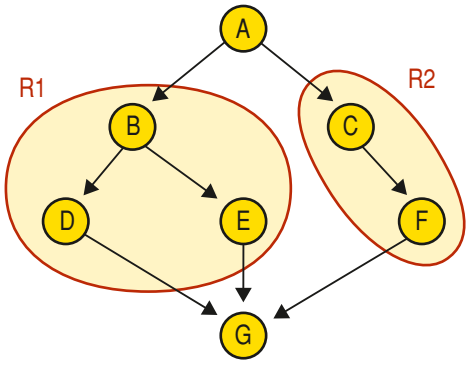
All'interno dei singoli processi possono a loro volta essere presenti delle istruzioni di cobegin-coend, cioè è possibile avere l'annidamento di più costrutti cobegin-coend.

ESEMPIO 12

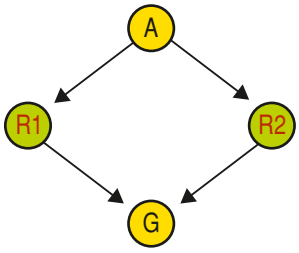
Codifichiamo il seguente grafo delle precedenze utilizzando il costrutto cobegin-coend.



Osservando il grafo delle precedenze possiamo notare come è sostanzialmente composto da due rami che possono essere evidenziati come nella seguente figura



Chiamando R1 e R2 le due sequenze di istruzioni, il grafo può essere semplificato come segue:



Il codice in pseudocodifica è quindi il seguente:

Processo P1	Processo R2	Processo R3

inizio
  A
  cobegin
  R1
  R2
  coend
  G
fine
	inizio
	  B
	  cobegin
	  D
	  E
	  coend
	fine
		inizio
		  C
		  F
		fine

Il costrutto cobegin-coend è decisamente più semplice da utilizzare e il codice che si ottiene è più strutturato e quindi più comprensibile di quello scritto con le istruzioni di fork-join che trasformano il codice in strutture che assomigliano ai vecchi programmi che utilizzavano l'istruzione di salto incondizionato "goto label", censurata dalla programmazione strutturata per la generazione di ◀ **"spaghetti code"** ▶ incomprensibili.

◀ **Spaghetti code** Spaghetti code is a derogatory term for computer programming that is unnecessarily convoluted, and particularly programming code that uses frequent branching from one section of code to another (using many GOTOs or other "unstructured" constructs). Spaghetti code sometimes exists as the result of older code being modified a number of times over the years. ▶

CAMMINO PARALLELO

Indichiamo con cammino parallelo una qualunque sequenza di nodi delimitata da un nodo COBEGIN e un nodo COEND.

Nell'esempio precedente è possibile individuare due cammini paralleli, uno esterno e uno annidato al suo interno.

Equivalenza di fork-join e cobegin-coend

Il costrutto fork-join può essere sostituito col costrutto cobegin-coend solo se non ci sono strutture annidate, nel tal caso l'unica possibilità di "conversione" è quella che ha la terminazione congiunta di tutti i processi: nel caso opposto, invece, sempre tutti i programmi codificati con cobegin-coend possono essere anche codificati con fork-join.

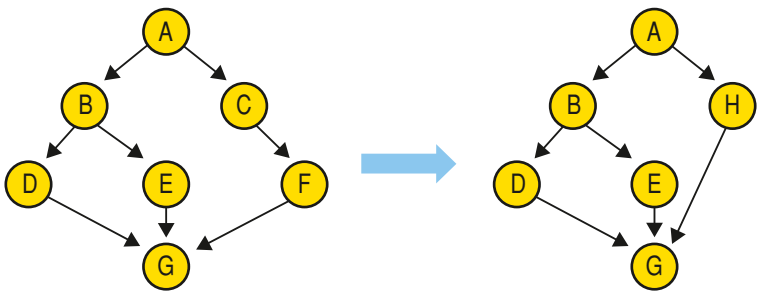
EQUIVALENZA TRA FORMALISMI

Qualunque programma parallelo può essere descritto utilizzando il costrutto fork-join, mentre non tutti i programmi paralleli possono essere descritti col costrutto cobegin-coend.

Dimostriamolo sperimentalmente tramite due semplici esempi: convertiamo un segmento descritto con cobegin-coend a fork-join e successivamente proviamo a eseguire l'operazione inversa.

ESEMPIO 13 ***Da cobegin-coend a fork-join***

Scriviamo mediante l'utilizzo di fork-join l'esempio che presenta due cobegin-coend annidati.

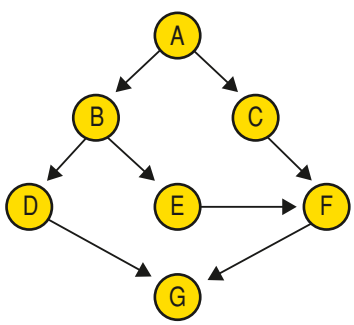


Dopo aver posto H = C+F, la pseudocodifica è la seguente:

```
/* processo padre: */
inizio
…
A
p2 = fork(H);          // inizia l'elaborazione parallela di B e H
   B
   p4 = fork(E);       // inizia l'elaborazione parallela di D e E
      D
   join p4;            // termina l'elaborazione parallela di D e E
join P2;               // termina l'elaborazione parallela del primo fork
…
fine.

/* processo figlio H */
inizio
   C
   F
fine
/* processo figlio E */
inizio
   E
fine
```

ESEMPIO 14 ***Da fork-join a cobegin-coend***

Ipotizziamo ora di dover descrivere mediante cobegin-coend la situazione rappresentata nel seguente grafo delle precedenze: ▶



Possiamo notare come il processo che esegue A si sospende per mandare in esecuzione due processi (cobegin P_B e P_C) dei quale il primo, a sua volta, si sospende per mandare in esecuzione altri due processi (cobegin P_D e P_E): è però impossibile determinare i corrispondenti coend in quanto i processi figli interagiscono tra di loro generando nuovi processi che non terminano assieme: quindi P_B non termina assieme a P_C

GRAFO STRUTTURATO

Un grafo per poter essere espresso soltanto con cobegin e coend deve essere tale che detti X e Y due nodi del grafo, tutti i cammini paralleli che iniziano da X terminano con Y e tutti quelli che terminano con Y iniziano con X (o, più sinteticamente, ogni 'sotto-grafo' deve essere del tipo one-in/one-out).
In questo caso il grafo si dice strutturato.

Il grafo dell'esempio precedente non presenta questa caratteristica e quindi, non essendo strutturato, costituisce un esempio impossibile da descrivere soltanto con cobegin e coend.

La codifica con il costrutto fork-join è invece fattibile, ma è necessario introdurre le istruzioni di salto in quanto le fork si intersecano tra loro:

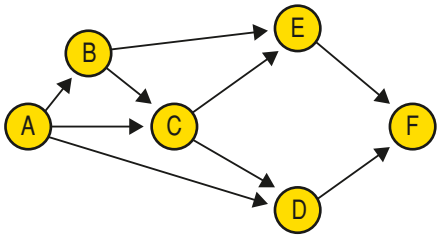
```
/* processo padre: */
inizio
…
A
p2=fork(C);        // inizia l'elaborazione parallela di B e C
   B
   P3=fork(E);     // inizia l'elaborazione parallela di D e E
      D
      goto L1
   join C;         // aspetta la terminazione di C
L1:join E2;        // aspetta la terminazione di E join C
G
…
fine.
```

La join etichettata con L1 non avviene tra gli stessi due processi che hanno fatto la prima fork, ma tra il primo processo e un processo generato dalla join tra due figli, il processo che esegue E e quello che esegue C.

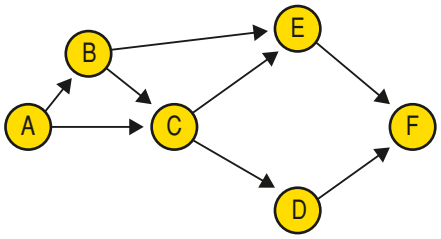
Semplificazione delle precedenze

Prima di affrontare la scrittura di un programma parallelo è sempre necessario soffermarsi ad analizzare il grafo delle precedenze per cercare di semplificarlo eliminando le precedenze implicite.

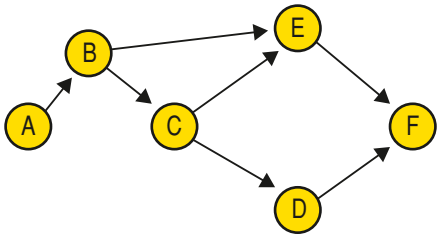
In presenza di precedenze implicite è possibile togliere un arco in quanto questo risulta ridondante ed è quindi possibile semplificare il grafo: vediamo un esempio e semplifichiamo il grafo riportato nella figura seguente:



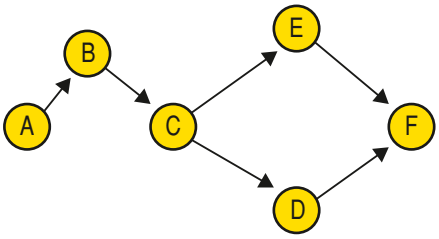
Possiamo osservare che il nodo D ha come precedenza il nodo A e anche il nodo C ha come precedenza il nodo A: quindi il nodo D ha il nodo A come precedenza diretta ma deve attendere l'elaborazione di C che a sua volta dipende da A; è possibile eliminare la precedenza tra A → D che risulta essere implicita in quella tra C → D: il grafo risulta essere trasformato nel seguente:



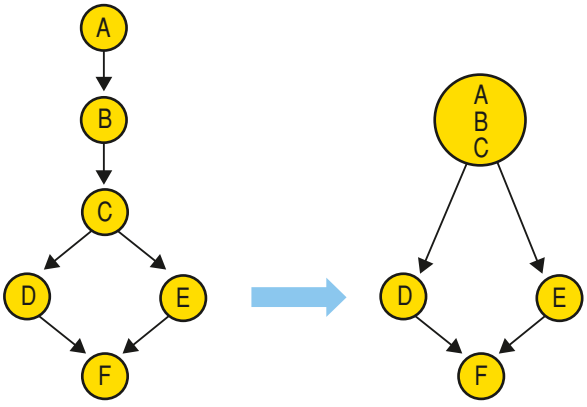
Analogo discorso può essere fatto tra B, che ha come precedenza il nodo A, e il nodo C, che anch'esso ha come precedenza il nodo A: quindi il nodo B ha il nodo A come precedenza diretta e dato che deve attendere l'elaborazione di B che a sua volta dipende da A è possibile eliminare la precedenza tra A → C che risulta essere implicita in quella tra B → C: il grafo risulta essere trasformato nel seguente:



Analogo discorso può essere fatto tra E, che ha come precedenza il nodo B, e il nodo C, che anch'esso ha come precedenza il nodo B: quindi il nodo E ha il nodo B come precedenza diretta e dato che deve attendere l'elaborazione di C, che a sua volta dipende da B, è possibile eliminare la precedenza tra B → E che risulta essere implicita in quella tra C → E; il grafo risulta essere trasformato nel seguente:



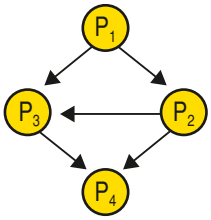
Ridisegnandolo, osserviamo che le tre operazioni A, B e C sono **sequenziali**, possono quindi essere raggruppate in un solo nodo, come si può vedere nel seguente disegno.



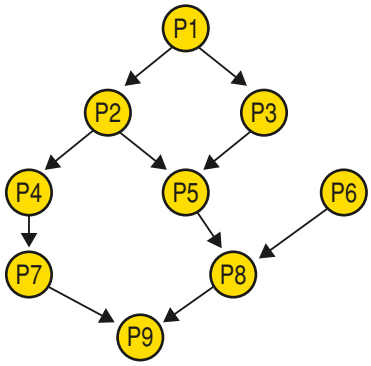
Il nostro grafo di partenza può essere trasformato in quest'ultimo, dove si vede che l'unica operazione che può essere parallelizzata è quella dei nodi D ed E: parallelizzare il primo grafo non porterebbe nessun vantaggio in termini di tempo di elaborazione.

Verifichiamo le competenze

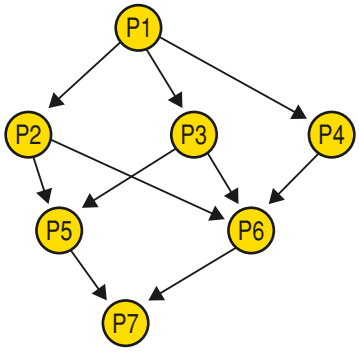
1 Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio utilizzando il costrutto fork-join.



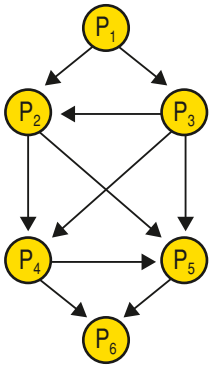
2 Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio utilizzando il costrutto fork-join.



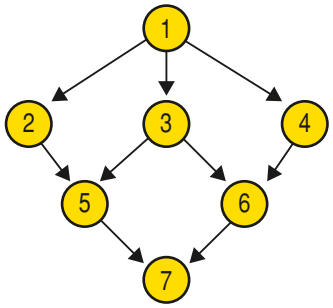
3 Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio utilizzando il costrutto fork-join.



4 Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio utilizzando il costrutto fork-join.



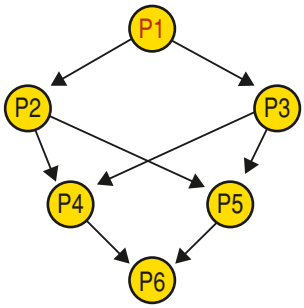
5 Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio utilizzando il costrutto fork-join.



6 Realizza un programma con cobegin-coend che implementi Mergesort.

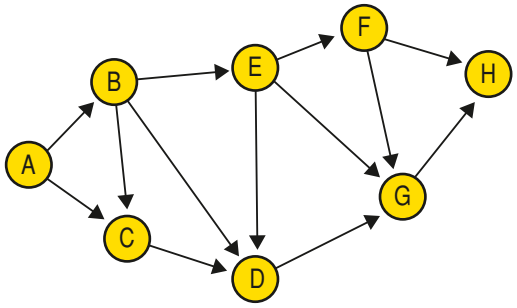
7 Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio:

a) utilizzando il costrutto fork-join
b) utilizzando il costrutto cobegin-coend



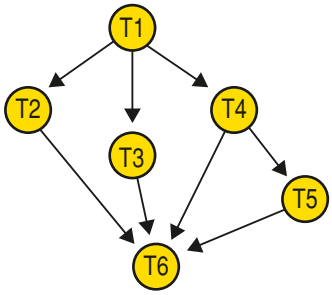
8 Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio:

a) utilizzando il costrutto fork-join
b) utilizzando il costrutto cobegin-coend

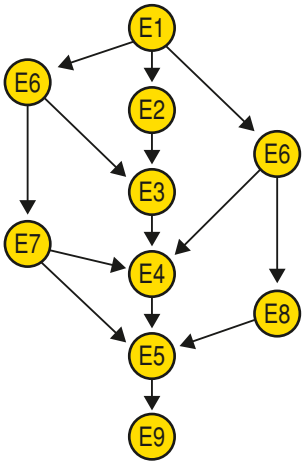


9 Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio:

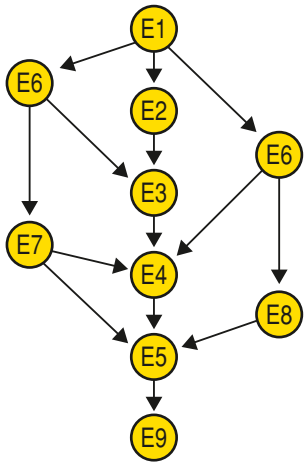
a) utilizzando il costrutto fork-join
b) utilizzando il costrutto cobegin-coend



10 Codifica in pseudolinguaggio il seguente grafo di precedenze utilizzando il costrutto fork-join.



11 Codifica in pseudolinguaggio il seguente grafo di precedenze utilizzando il costrutto fork-join (si suggerisce di semplificarlo prima di effettuare la codifica).



12 Partendo dal diagramma delle precedenze realizzato con il codice dell'esercizio 5 della lezione 4:

a) sviluppa il programma utilizzando il costrutto COBEGIN-COEND
b) sviluppa il programma utilizzando il costrutto FORK-JOIN

13 Partendo dal diagramma delle precedenze realizzato con il codice dell'esercizio 6 della lezione 4:

a) sviluppa il programma utilizzando il costrutto COBEGIN-COEND
b) sviluppa il programma utilizzando il costrutto FORK-JOIN

14 Partendo dal diagramma delle precedenze realizzato con il codice dell'esercizio 7 della lezione 4:

a) sviluppa il programma utilizzando il costrutto COBEGIN-COEND
c) sviluppa il programma utilizzando il costrutto FORK-JOIN

Esercitazioni di Laboratorio 1

L'EMULATORE CYGWIN

Cos'è Cygwin

La più concisa (e al tempo stesso esauriente) spiegazione di cosa sia ◀ **Cygwin** ▶ la si trova sul sito Internet dove esso viene distribuito (http://www.cygwin.com):
Cygwin permette di avere su una macchina Windows la potenza e la flessibilità di una shell ◀*NIX Like ▶ e quindi di utilizzare molti dei software tipici del mondo Linux.

◀ **Cygwin** *"Cygwin is a Linux-like environment for Windows. It consists of two parts:*
◗ *A DLL (cygwin1.dll) which acts as a Linux API emulation layer providing substantial Linux*
◗ *API functionality.*
A collection of tools which provide Linux look and feel." ▶

◀ *NIX Like Con *NIX si intendono i sistemi operativi UNIX e XENIX e con *NIX Like si intendono i sistemi operativi che sono loro "somiglianti", come Linux che sappiamo essere direttamente derivato da UNIX. ▶

Il vantaggio di utilizzare un ambiente emulato invece di uno "nativo" consiste nel non dover installare sulla propria macchina un nuovo sistema operativo, con tutte le problematiche che questa operazione solitamente comporta.

Naturalmente è diverso avere a disposizione una macchina Linux vera e una emulata, soprattutto in termini di risorse in quanto l'emulatore occupa buona parte della memoria RAM disponibile, ma per i programmi che verranno scritti durante questo corso, molto piccoli e "leggeri", l'ambiente emulato non si differenzia dall'ambiente reale, sia nel linkaggio che nella compilazione e nell'esecuzione.

Perché istalliamo Cygwin

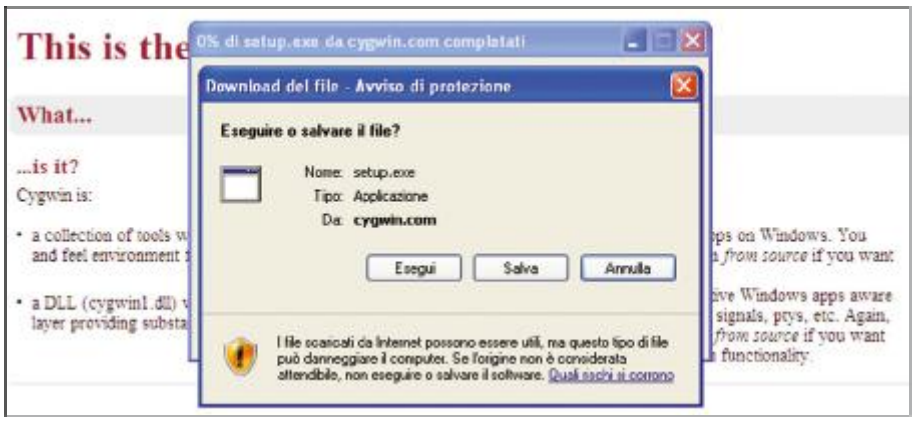
Naturalmente se stiamo lavorando su una macchina *NIX Like o ne abbiamo una partizione sul nostro PC non abbiamo bisogno di istallare questo programma.
Ma per eseguire una parte dei programmi concorrenti, in particolare quelli che eseguono la *fork*, è necessario avere a disposizione un sistema operativo *NIX like: durante l'istallazione di Cygwin scaricheremo anche il compilatore C (il compilatore GCC), in modo da avere a disposizione "sotto Windows" un emulatore di macchina Linux completa, in modo da per effettuare le nostre prove.

Installare Cygwin

Per installare Cygwin è necessario utilizzare un computer connesso a Internet: nella home page del sito http://www.cygwin.com si può individuare il link all'installer di Cygwin (http://www.cygwin.com/setup.exe, contraddistinto dall'etichetta *"setup.exe"*.

Nel momento in cui scrivo questa esercitazione, la versione di Cygwin più recente è quella comprendente la DLL in versione 1.7.16-1. Le istruzioni che seguono faranno dunque riferimento all'installazione di questa specifica versione.

Un semplice click con il pulsante sinistro del mouse su questo collegamento e una finestra come quella riportata di seguito chiederà che cosa si desidera fare con il file "setup.exe" cui si sta cercando di accedere.

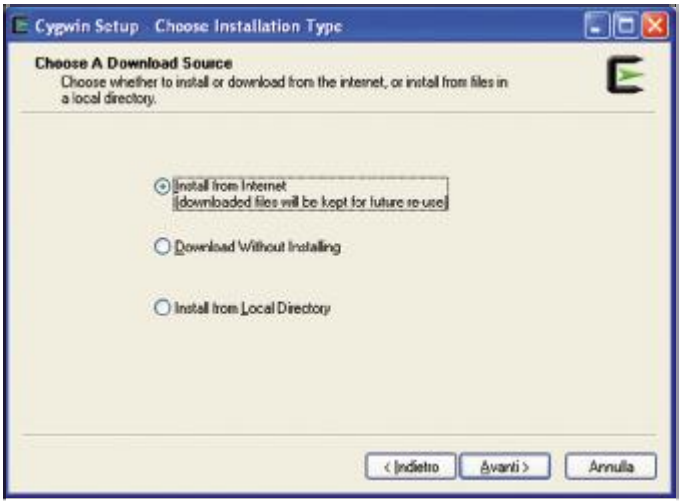


Scegliamo di salvare il file (sono circa 250kb) e posizioniamolo sul Desktop oppure in una directory a piacere (nel mio caso ho selezionato la directory Dev-Cpp).

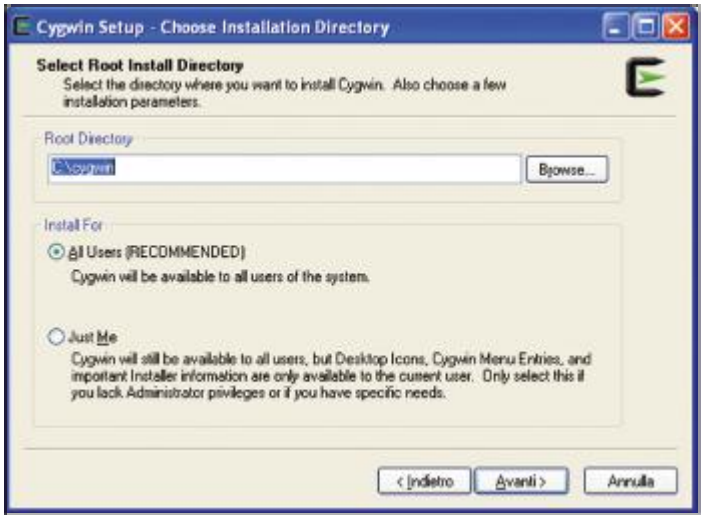
Ultimato il download autorizziamo l'esecuzione del programma di istallazione cliccando Esegui nella finestra di protezione che ci si presenta davanti. ▶

E successivamente mandiamo in esecuzione l'installer cliccando su cliccando Avanti ▶

Alla successiva finestra confermiamo quanto ci viene proposto, cioè la prima opzione ("*Install from Internet*") e clicchiamo il pulsante Avanti per continuare. ▶

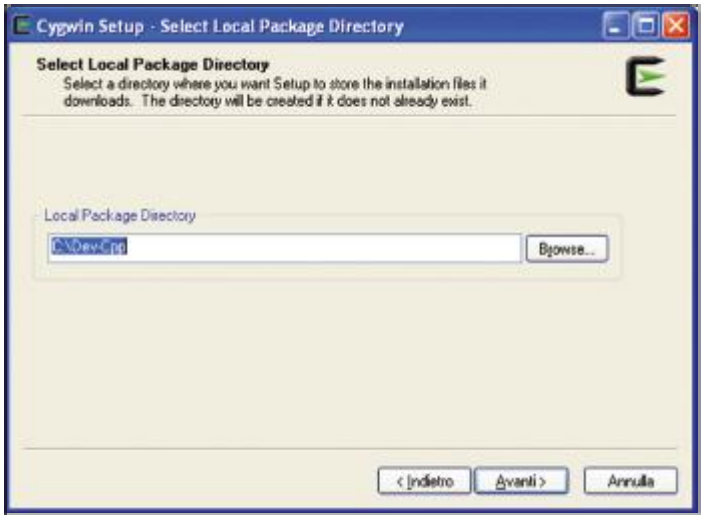


È necessario indicare in quale directory installare Cygwin: questa cartella sarà in seguito la root del nostro sistema e quindi deve essere facilmente raggiungibile; si consiglia di lasciare quella da lui proposta di default, cioè "*C:\cygwin*" ▶

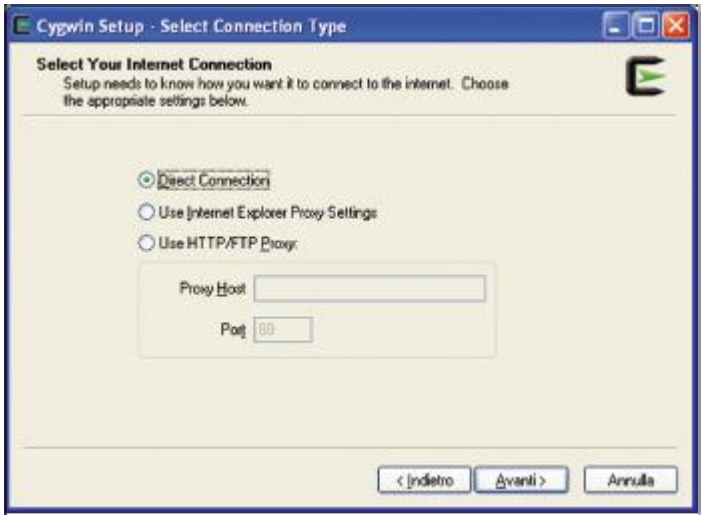


Senza modificare l'altra impostazione ("*Install For → All Users*") clicchiamo sul pulsante Avanti per continuare.

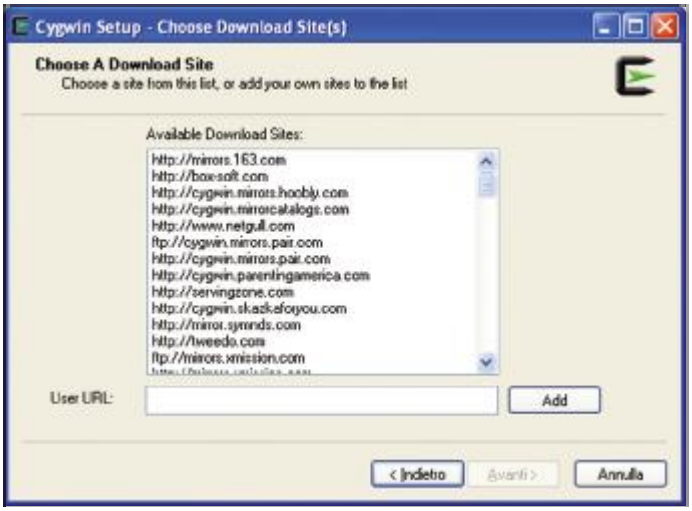
L'installer di Cygwin, a questo punto, ci chiede dove vogliamo copiare i file che vengono scaricati per il download, proponendoci la conferma della directory dove abbiamo copiato il file di setup.
Confermiamo e proseguiamo sempre cliccando sul pulsante Avanti. ▶



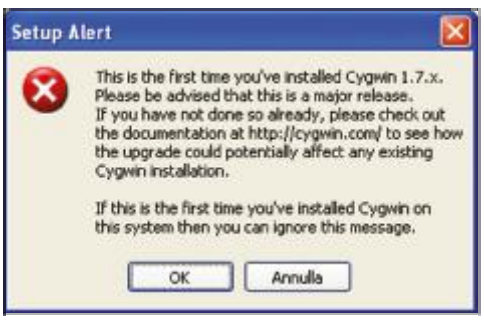
Selezioniamo ora come sarà effettuato il download: in caso di normale connessione via modem si conferma quanto proposto e si procede sempre cliccando sul pulsante Avanti. ▶



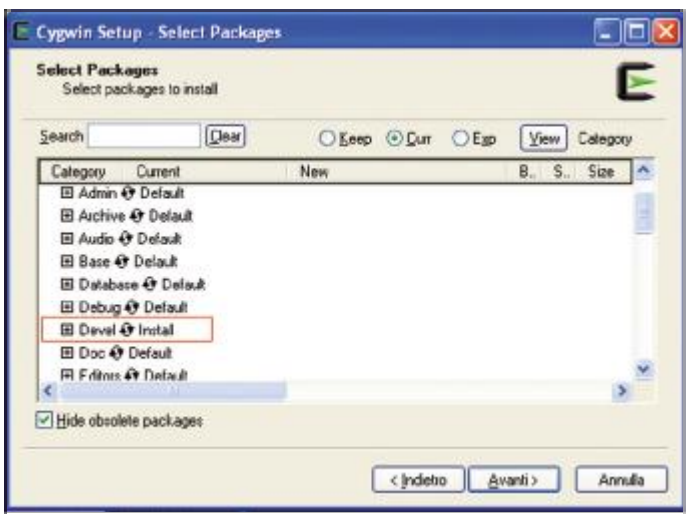
Dobbiamo ora selezionare un server da cui scaricare i file: scelto uno a piacere (si consiglia uno europeo) si prosegue sempre cliccando sul pulsante Avanti.



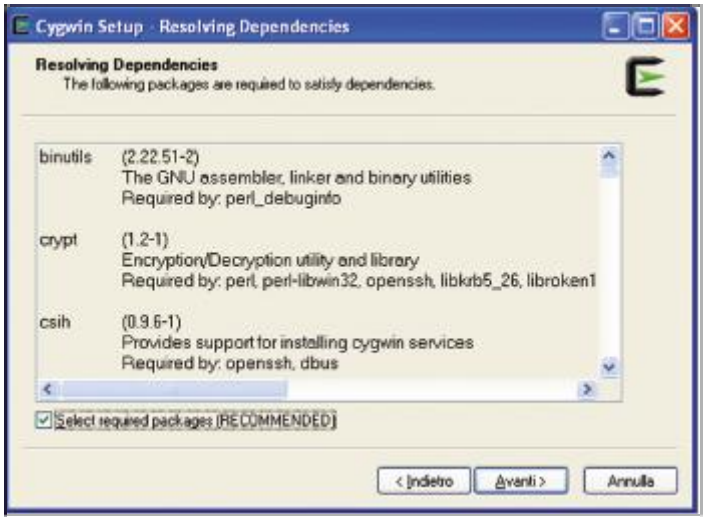
Il sistema si accorge che è la prima volta che installiamo Cygwin e ci consiglia di effettuare l'installazione della documentazione. ▶



La successiva schermata ci permette di selezionare i pacchetti che intendiamo istallare: per utilizzare il compilatore GCC è necessario modificare le impostazioni di default e includere il "kit" per gli sviluppatori (developers) cliccando su Devel in modo che a fianco sia presente *Install*", come nella seguente immagine.



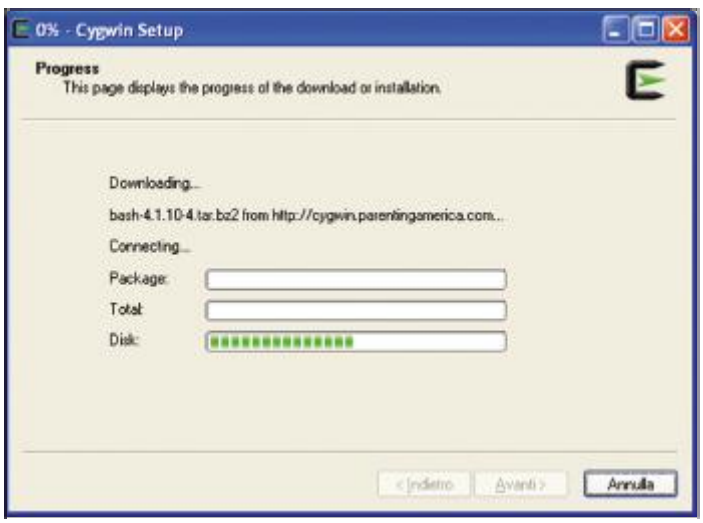
Allo stesso modo si può selezionare il dowload della documentazione, abilitando nella riga successiva Doc: si prosegue cliccando sul pulsante Avanti.



Se si presenta la seguente videata si prosegue cliccando nuovamente sul pulsante Avanti. ▶

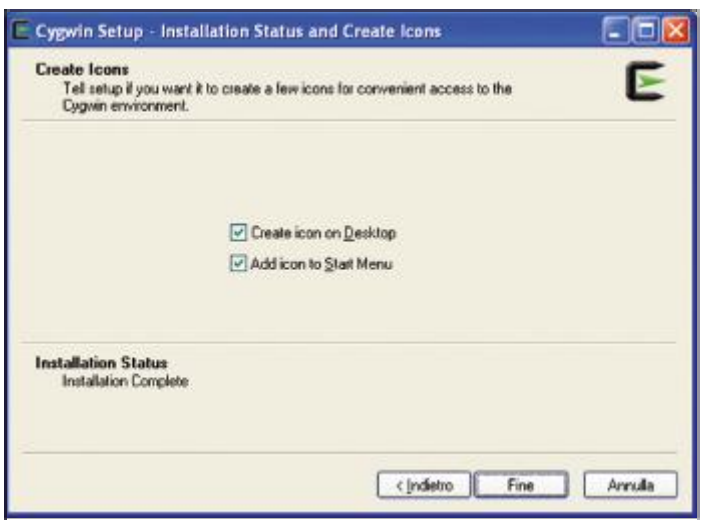
Nulla da modificare nella schermata seguente sempre cliccando sul pulsante Avanti.

Inizia ora il download e l'istallazione vera e propria: sullo schermo ora è presente la seguente videata: ▶



> L'istallazione completa dell'emulatore, che ha dimensioni di circa 3,62 GB, richiede dai 20 ai 40 minuti, a seconda della velocità della connessione.

Al termine dell'installazione di Cygwin viene visualizzata la seguente schermata che chiede dove posizionare i collegamenti per lanciare l'applicazione: accettiamo e confermiamo cliccando su Fine. ▶



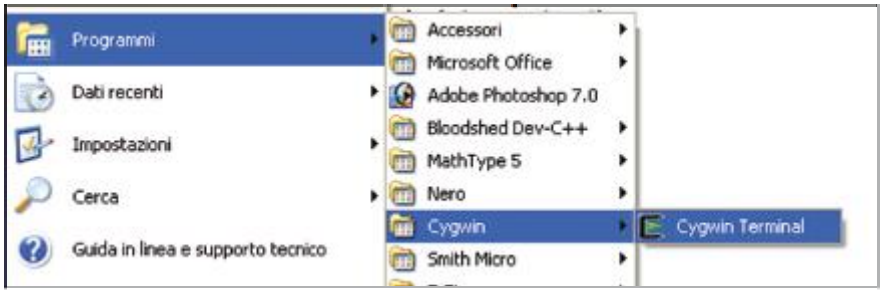
Siamo ora pronti a utilizzare l'emulatore, a scrivere i programmi e compilarli con GCC.

Il primo programma in GCC

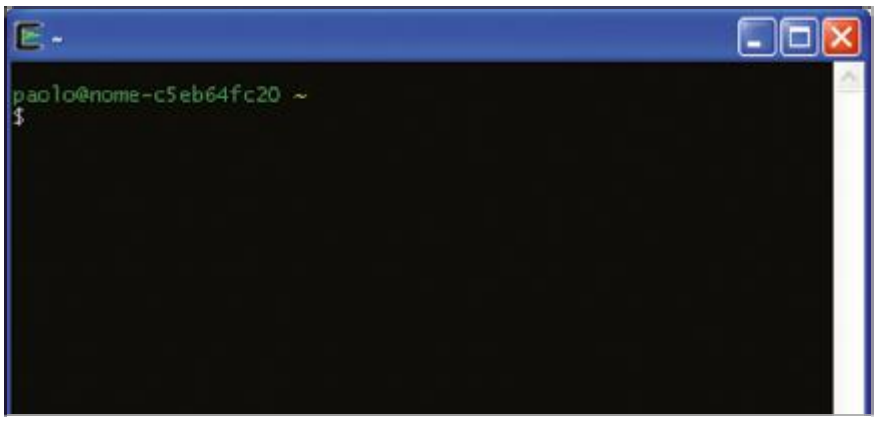
Ora che abbiamo il nostro ambiente di esecuzione, proviamo a scrivere il nostro primo programma, a compilarlo con GCC e a mandarlo in esecuzione.

Per prima cosa avviamo l'emulatore cliccando sull'icona presente sul desktop: ▶

oppure dal menu di avvio, nell'opzione Programmi.



Ci si presenta una finestra nera, come quella di figura, tipica dello shell dei comandi degli ambienti testuali (come il DOS).



Il nome che viene proposto è il *nome_utente* assegnato in Windows dell'utente che ha installato Cygwin.

Diamo alcuni comandi, giusto per impratichirci dell'ambiente: consigliamo di scaricare un tutorial che spieghi i comandi della shell Linux.

Il primo comando che proviamo è quello che ci permette di muoverci nell'albero delle directory, cioè il comando *cd*: spostiamoci dalla nostra directory alla radice digitando

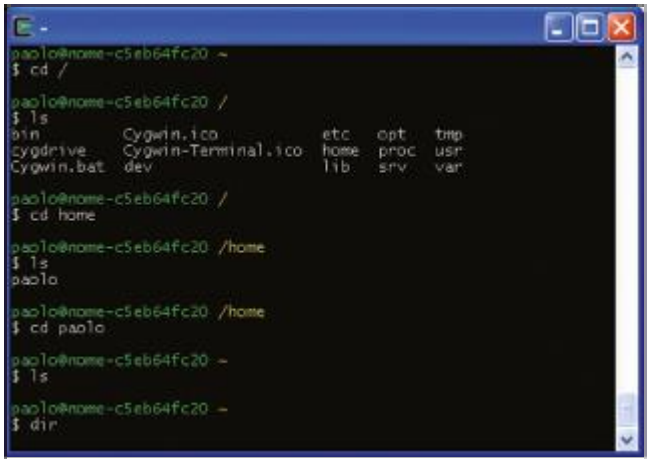
```
cd /
```

Quindi ne visualizziamo il contenuto col comando

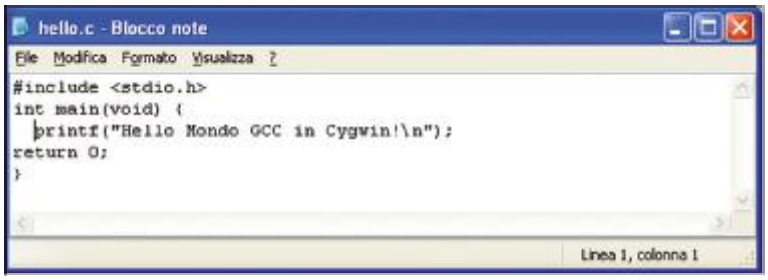
```
ls /
```

Entriamo poi nella directory home col comando *cd home* e ne visualizziamo il contenuto: è presente la sola cartella col nostro nome, ci entriamo e visualizziamo l'elenco dei file presenti sempre con *ls* (oppure anche con il comando *dir*).

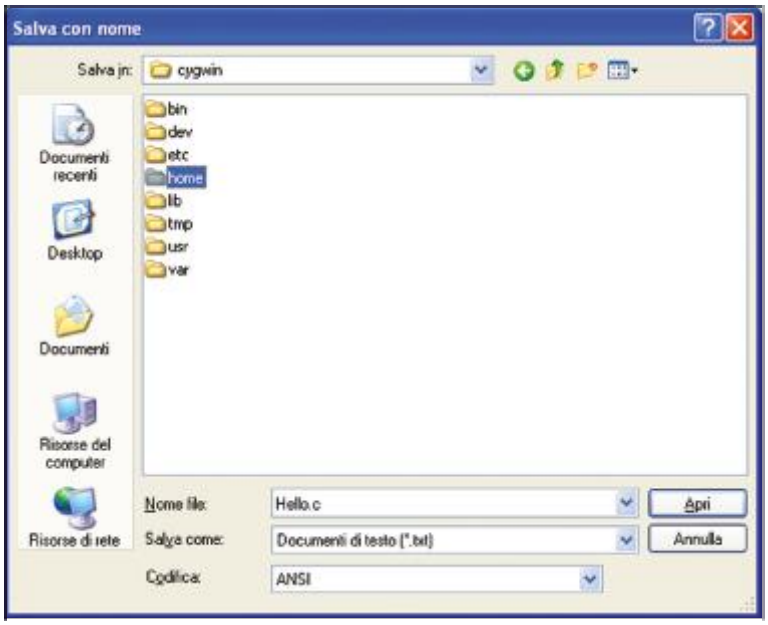
In figura possiamo vedere l'esito dei comandi sopra elencati. ▶



Per scrivere il codice C è necessario usare un editor: si può usare il semplice blocco note, come nell'esempio di figura: ▶



… e salvare il file nella cartella personale (nel mio caso paolo) all'interno della subdirectory home della sotto cartella nella quale abbiamo istallato Cygwin, nel nostro caso "*C:\cygwin*": ▶



e salviamolo con il nome Hello.c.

È anche possibile utilizzare come editor Dev-cpp, scaricabile gratuitamente all'indirizzo http://www.bloodshed.net/devcpp.html oppure dalla cartella materiali nella sezione riservata al presente volume sul sito www.hoepliscuola.it.

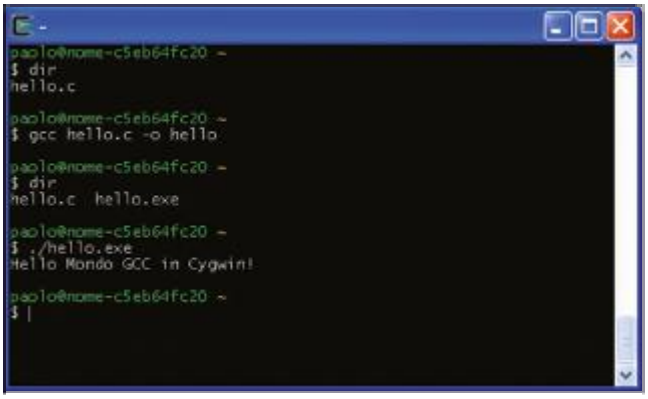
Se ora digitiamo dir al prompt dei comandi di Cygwin troviamo il nostro file sorgente pronto per essere compilato con GCC: lo compiliamo col comando

```
gcc Hello.c –o hello
```

Digitando nuovamente dir al prompt ora visualizzeremo due file: oltre al file sorgente troviamo il file eseguibile che mandiamo in esecuzione col comando:

```
./hello.exe
```

L'output e la sequenza dei comandi fin qui elencati è riportata nella successiva videata:



Abbiamo realizzato il nostro primo programma in C in ambiente Cygwin utilizzando il compilatore GCC!

GCC è un tool ricco di opzioni. Il manuale ufficiale lo si può trovare all'indirizzo: http://gcc.gnu.org/onlinedocs/.
Per esempio, per conoscere la versione di GCC che si sta utilizzando è possibile utilizzare il comando gcc –-ver.

Prova adesso!

I programmi concorrenti che scriveremo necessitano di un insieme di librerie, riportate nella seguente figura che andremo ad aggiungere all'inizio di tutti i nostri codici C.

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <sys/wait.h>
4 #include <fcntl.h>
5 #include <stdio.h>
6 #include <unistd.h>
7 #include <stdlib.h>
```

Scriviamo ora il primo programma che genera un processo figlio eseguendo una operazione di fork.

L'istruzione

```
fork()
```

Ha come parametro di ritorno un valore integer che nel processo padre è il PID del processo appena creato mentre nel processo figlio è uguale a 0.
Cioè, dato che dopo l'esecuzione della fork vengono eseguite parallelamente le istruzioni da padre e figlio, se inseriamo un'istruzione di selezione

```
if (pid=0)
```

darà risultato VERO se è il processo figlio a eseguirla mentre darà risultato FALSO se è il processo padre che la esegue: quindi dopo questo test nei due rami della selezione possiamo inserire le istruzioni che devono essere eseguite o dal padre o dal figlio.
Il codice completo è il seguente:

```
1  #include <sys/types.h>
2  #include <sys/stat.h>
3  #include <sys/wait.h>
4  #include <fcntl.h>
5  #include <stdio.h>
6  #include <unistd.h>
7  #include <stdlib.h>
8
9  void main(){
10   pid_t pid;
11   printf("1) prima della fork \n");
12   pid=fork();
13   printf("2) dopo della fork \n");
14   if (pid==0){
15     printf("sono il processo figlio\n");
16     exit(1) ;      /* termina processo figlio */
17   }else{
18     printf("sono il processo padre\n");
19     exit(0) ;      /* non necessaria */
20   }
21 }
22
```

1. Modifica ora il programma facendo il test non sul pid = 0 ma confrontandolo col il valore che viene generato alla sua creazione, in modo da "invertire" i due rami della selezione.
2. All'interno dei due rami, oltre alla frase di saluto visualizza anche il valore del pid.
3. Scrivi un nuovo programma dove vengono generati tre processi e fai scrivere a ciascuno di essi sullo schermo rispettivamente "Ciao, io sono Qui", "Ciao, io sono Quo" e "Ciao, io sono Qua".

Esercitazioni di Laboratorio 2

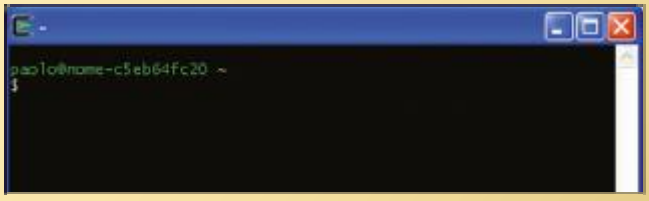
L'AMBIENTE DI SVILUPPO DEV-C++

Ambiente di sviluppo e di esecuzione

Nella precedente esercitazione di laboratorio abbiamo installato l'emulatore Cygwin completo di compilatore C (il compilatore GCC) in modo da avere a disposizione "sotto" Windows una "macchina Linux" completa per effettuare le nostre prove.

La scelta didattica da noi effettuata per compilare ed eseguire i programmi in linguaggio C è stata quella di utilizzare proprio tale compilatore all'interno dell'ambiente operativo offerto da Cygwin, in modo da permettere di effettuare le esercitazioni anche a coloro che hanno una macchina Windows e non hanno una partizione Linux sulla propria macchina ma che, semplicemente e in pochi passaggi, possono istallare tale emulatore.

I programmi che verranno presentati sono stati collaudati nel Terminal di Cygwin riportato a fianco. ▶



Questo ambiente, purtroppo, non mette a disposizione un editor evoluto oppure un ambiente di sviluppo per il linguaggio C: sfruttando però l'ambiente multitasking proprio di windows possiamo utilizzare i prodotti di sviluppo scritti per il sistema operativo windows, salvare i file sorgenti all'interno delle cartelle dell'emulatore e quindi compilare e mandare in esecuzione il programma così realizzato dalla linea di comando.

Noi utilizziamo a tale scopo l'ambiente Dev-C++, che è un ambiente di sviluppo integrato (IDE) completo per la programmazione sia in linguaggio C++ che in linguaggio C sotto Windows, prodotto dalla Bloodshed con licenza GNU (cioè di software completamente libero e gratuito) e include:

- un potente editor multi-finestra con varie opzioni;
- un compilatore C++ (e quindi anche C) per Windows;
- un debugger;
- un evidenziatore di sintassi personalizzabile e molto altro ancora (non è un caso se questo software è usato per le *Olimpiadi Mondiali dell'Informatica*).

Abbiamo scelto di utilizzare l'ambiente C++ piuttosto che quello del C in quanto questo ambiente è disponibile in lingua italiana.

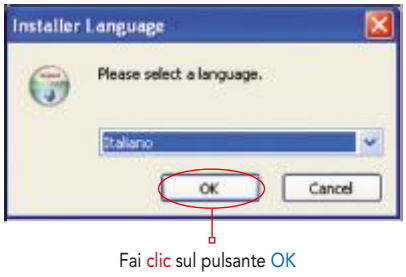
Installiamo l'ambiente di lavoro Dev-C++

Per poter utilizzare l'ambiente di lavoro Dev-C++ è innanzitutto necessario installare questo programma sul personal computer, attraverso la seguente procedura.

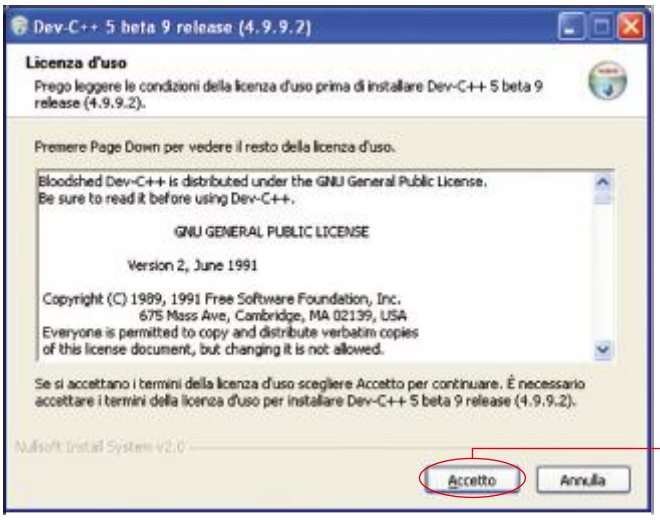
1. Crea nella directory principale del tuo hard-disk la cartella C_Hoepli dove copierai i materiali relativi al linguaggio C.
2. Scarica nella cartella materiali della sezione riservata a questo volume del sito www.hoepliscuola.it il programma di installazione devcpp-4.9.9.2_setup e copialo nella cartella C_Hoepli che hai appena creato sul tuo disco.
3. Per effettuare l'installazione è sufficiente effettuare un doppio clic sull'icona corrispondente.
4. Nel caso venisse visualizzato sullo schermo un avviso di protezione simile a quello della figura sottostante, fai clic sul pulsante Esegui.

5. Successivamente viene visualizzata sullo schermo la seguente raccomandazione, in cui si chiede di non installare questa versione di Dev-C++ sopra una precedente installazione già presente sul disco fisso.

6. Seleziona la lingua desiderata per la procedura di installazione.

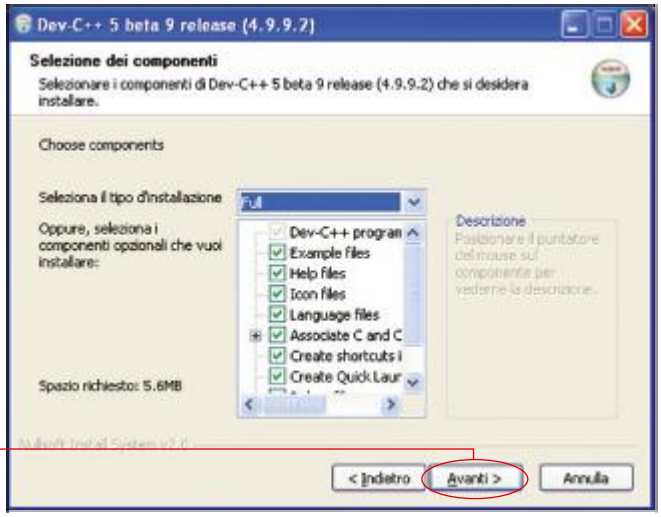


7. Le successive tre schermate ti permettono di completare l'installazione del programma. La prima schermata richiede la conferma di accettazione della licenza GNU. Procedi facendo clic sul pulsante Accetto.



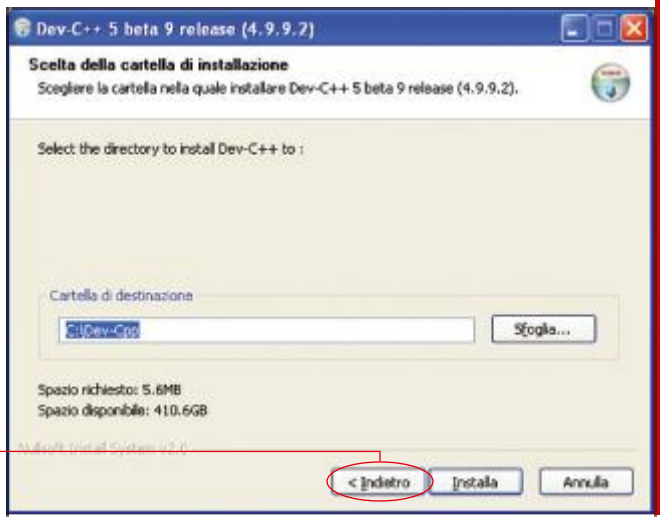
Fai clic sul pulsante Accetto

La seconda schermata ti permette di personalizzare il tipo di installazione scegliendo i componenti che verranno copiati nel calcolatore: lascia inalterato quanto proposto e procedi facendo clic sul pulsante Avanti. ▶



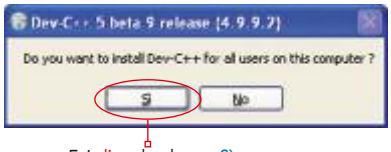
Fai clic sul pulsante Avanti

La terza schermata ti permette di personalizzare la directory in cui verrà copiato il programma (si consiglia di non effettuare modifiche a questa proposta).
Procedi facendo clic sul pulsante Installa. ▶



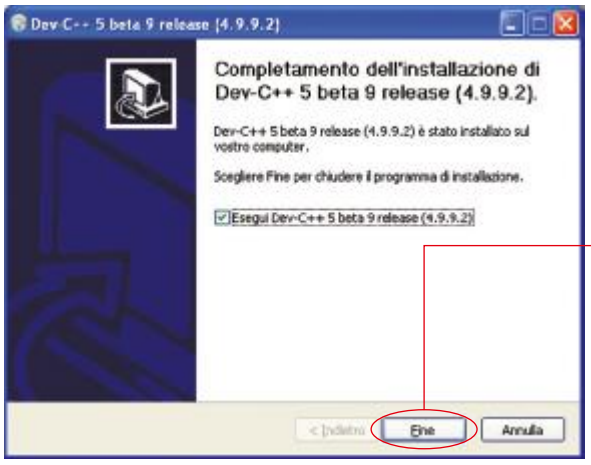
Fai clic sul pulsante Installa

8 Al termine dell'installazione dei file, sul disco rigido viene chiesto se l'ambiente Dev-C++ può essere usato da tutti gli utenti del computer: rispondi Sì e prosegui. ▶



Fai clic sul pulsante Sì

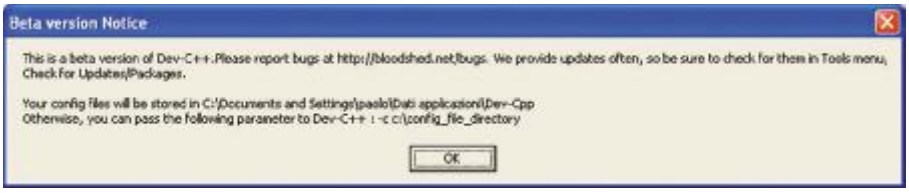
9 Se l'installazione si è conclusa con successo viene visualizzata sullo schermo la videata sottostante: puoi chiuderla facendo clic sul pulsante Fine.



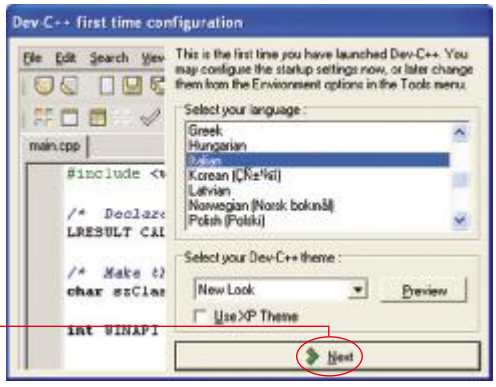
Fai clic sul pulsante Fine

Prima di iniziare a usare il programma è tuttavia necessaria una ulteriore fase di configurazione.

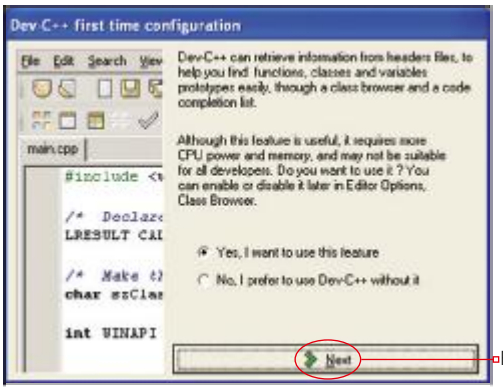
10 Dapprima ti viene segnalato dove è possibile reperire le informazioni su eventuali bug presenti nell'ambiente (la versione a disposizione è infatti ancora in fase beta, cioè in fase di collaudo: per l'utilizzo che ne viene fatto in questa sede abbiamo comunque potuto verificare che, di fatto, essa è perfettamente funzionante) e dove viene salvato il file di configurazione sul nostro PC.



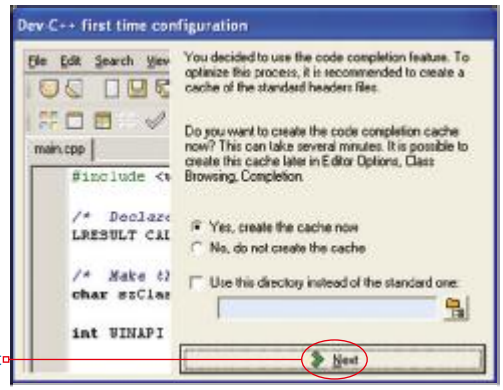
11 Facendo clic sul pulsante OK appare una finestra in cui viene richiesta la lingua preferita per l'intestazione dei menu dell'ambiente: dopo averla selezionata, fai clic sul pulsante Next nelle successive tre videate per accettare la configurazione di default proposta dal produttore. ▶



Fai clic sul pulsante Next

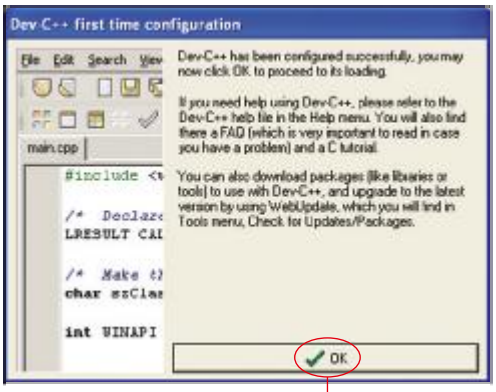


Fai clic sul pulsante Next



⓬ L'ultima schermata di installazione ti comunica che tutto è andato a buon fine e, facendo clic su OK, viene caricato l'ambiente Dev-C++. ▶

A questo punto appare (figura a lato) la schermata completa dell'◀ ambiente Dev-C++ ▶ e siamo pronti per scrivere il nostro primo programma.



Fai clic sul pulsante OK

◀ L'ambiente che abbiamo installato comprende sia il C sia il C++, è cioè in grado di compilare programmi scritti sia in C sia in C++: il C++, infatti, utilizza come linguaggio base la stessa sintassi del linguaggio C e tutte le istruzioni che studieremo sono "comuni" a entrambi i linguaggi. ▶

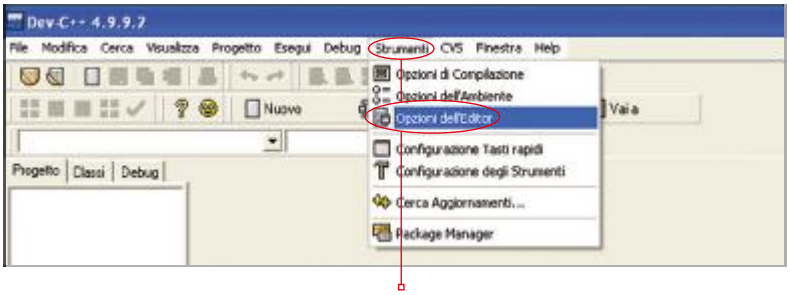
A ogni apertura del compilatore viene proposto al programmatore un suggerimento che favorisce l'apprendimento dell'ambiente di sviluppo: nel caso in cui tali aiuti non fossero graditi, è sufficiente inserire un segno di spunta nel check box in basso a sinistra della finestra.

Inserisci la spunta in Non mostrare i suggerimenti all'avvio

Scriviamo il nostro primo programma

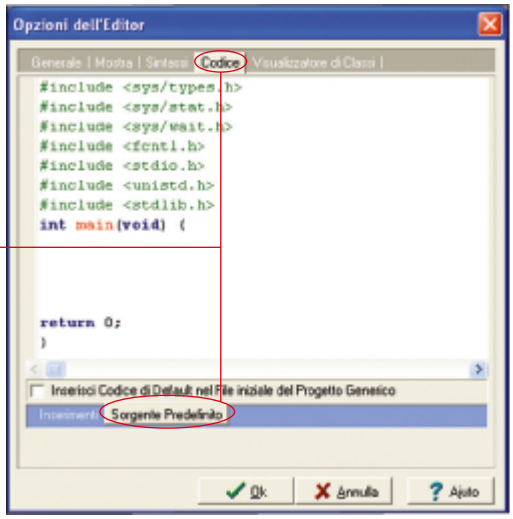
Innanzitutto dobbiamo effettuare la personalizzazione dell'ambiente di lavoro, in modo tale da essere successivamente agevolati nella scrittura dei programmi: è infatti possibile fare in modo che l'ambiente Dev-C++ proponga automaticamente "parte del programma" già scritto all'inizio di ogni nostro lavoro.

1. Per ottenere questo risultato è sufficiente preimpostare le istruzioni che desideriamo trovare già scritte scegliendo, dal menu Strumenti, Opzioni dell'Editor.



Seleziona Strumenti e, dal menu a tendina, scegli Opzioni dell'Editor

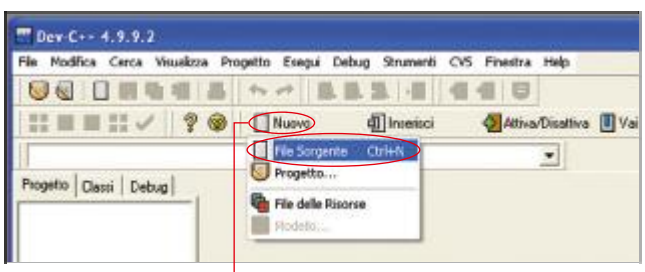
2. Nella finestra che appare selezioniamo la scheda Codice e successivamente la scheda Sorgente Predefinito, e scriviamo, per esempio, l'elenco delle librerie necessarie per l'esecuzione dei nostri programmi, come mostrate nella schermata a lato. ▶



Seleziona la scheda Codice...
... quindi seleziona la scheda Sorgente Predefinito

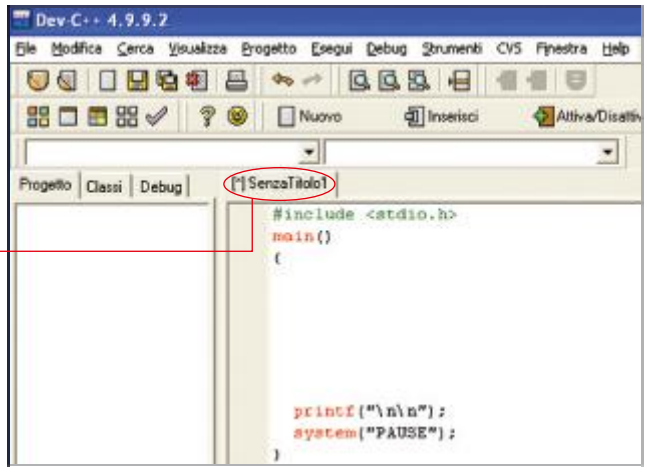
Queste righe verranno sempre scritte automaticamente in ogni nostro nuovo programma: successivamente potremo modificare questa impostazione per aggiungere anche altre istruzioni, come per esempio commenti di intestazione per la documentazione del programma.

3. Facendo clic su OK si ritorna al menu principale, dove possiamo iniziare a scrivere il nostro primo programma: a tal fine, ci posizioniamo sul pulsante Nuovo e, nel menu a tendina che ci appare, selezioniamo la voce File Sorgente. ▶



Posizionati su Nuovo e scegli File Sorgente

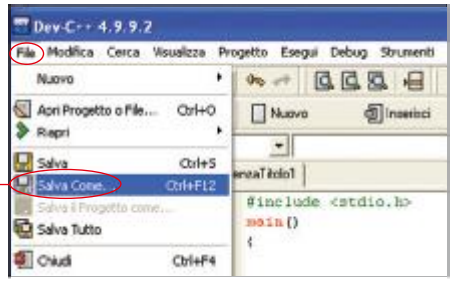
4 Viene visualizzata la finestra di editor del nostro primo programma, in cui sono presenti le righe definite in precedenza nella personalizzazione. ▶



Nome del programma

Come possiamo vedere dall'immagine, il programma si chiama SenzaTitolo1: diamo allora un nome al nostro "programma" salvandolo su disco, in un file che chiameremo programma1.c.

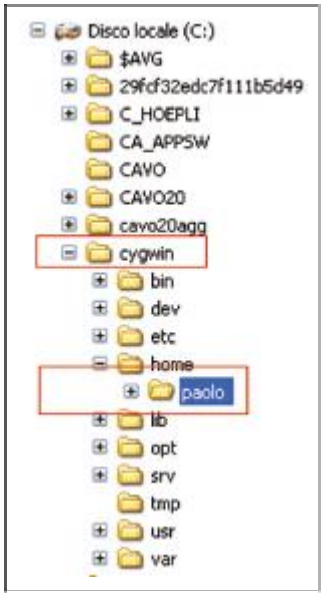
5 Per poter nominare il file, selezioniamo, dal menu File, l'opzione Salva Come… ▶



Seleziona File e, dal menu a tendina, scegli Salva Come...

6 Nella videata che appare, per poter attribuire il nome al file, è necessario, nell'ordine:

- scorrere il nostro disco per individuare la cartella dell'ambiente cygwin e la cartella di lavoro della nostra utenza (nel mio caso la directory paolo): ▶
- in Nome file, rinominare il programma come programma1;
- selezionare, in Salva come, il formato C.



Modifichiamo ora il codice, scrivendo tra le righe già predisposte la seguente istruzione:

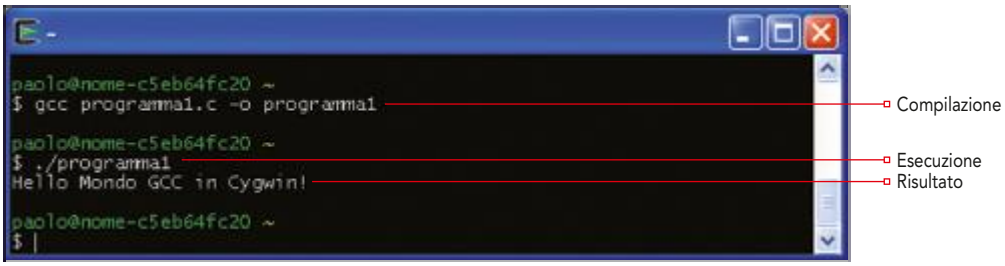
```
printf("Hello Mondo GCC in Cygwin!\n");
```

Otteniamo così la situazione riportata nella figura a lato. ▶

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <sys/wait.h>
4 #include <fcntl.h>
5 #include <stdio.h>
6 #include <unistd.h>
7 #include <stdlib.h>
8 int main(void) {
9
10  printf("Hello Mondo GCC in Cygwin!\n");
11
12
13 return 0;
14 }
```

Per compilarlo ci portiamo ora nell'ambiente cgwin cha mandiamo in esecuzione direttamente dal menu di avvio, nell'opzione Programmi.

Nella finestra dello shell dei comandi effettuiamo la compilazione e successivamente mandiamo in esecuzione il nostro programma eseguibile ottenendo sullo schermo il risultato della elaborazione.



Prova adesso!

Utilizzando Dev-C++ scrivi un nuovo programma dove vengono generati due processi e successivamente a ciascuno di essi fai generare un nuovo processo. Quanti sono in tutto i processi generati?

A rispondi prima simulando l'esecuzione sulla carta;
B scrivi il programma C e verifica la tua risposta praticamente.

Esercitazioni di Laboratorio 3

LA FORK-JOIN IN C

Info

I codici sorgenti sono nel file C_fork.rar scaricabile dalla cartella materiali nella sezione del sito www.hoepliscuola.it riservata al presente volume.

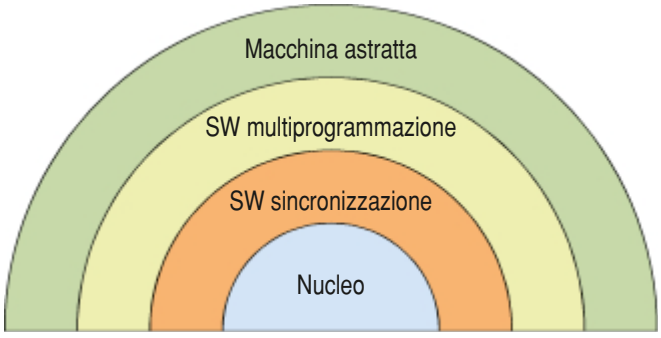
Macchina astratta

Per scrivere e collaudare programmi concorrenti è necessario avere a disposizione una macchina concorrente in grado di eseguire più processi sequenziali contemporaneamente e un linguaggio di programmazione con il quale descrivere algoritmi non sequenziali.

Il linguaggio C permette di descrivere programmi composti da un insieme di processi sequenziali asincroni interagenti e in questa esercitazione analizzeremo come scrivere e collaudare processi paralleli, cioè algoritmi che per loro natura hanno una sequenza delle attività con ordinamento parziale, dove l'esecutore per alcune istruzioni "è libero" di scegliere quali iniziare prima senza che il risultato sia compromesso: può quindi anche "portare avanti" l'elaborazione di questi segmenti di codice in parallelo.

Il nostro esecutore, essendo un normale personal computer, non ha tante unità di elaborazione quanti sono i processi da svolgere e quindi realizzerà una macchina concorrente astratta con tecniche software o direttamente grazie alle primitive del sistema operativo.

Quindi nei sistemi monoprocessore la macchina parallela è una macchina astratta che possiamo vedere strutturata nei seguenti livelli: ▶



Il nucleo corrisponde all'esecutore del programma compilato in linguaggio concorrente che dispone di un insieme di meccanismi di sincronizzazione che permettono la realizzazione di applicazioni multiprogrammate.

In questa unità esercitazione implementeremo i costrutti fondamentali descritti ad alto livello nella lezione 5 della presente unità di apprendimento:

- il costrutto fork-join;
- il costrutto cobegin-coend.

Il costrutto fork-join in linguaggio C: le istruzioni fork() e wait()

In Linux tutte le operazioni vengono svolte dai processi che possono generare ulteriori processi (*child process*): ogni processo ha un padre, tranne il processo di partenza, che in generale è *init*.

> Se un processo figlio "perde" il padre per qualunque motivo diviene anch'esso figlio di *init*.

Le istruzioni fork e join di Dennis e VanHorne sono realizzate in linguaggio C mediante due funzioni:

Fork

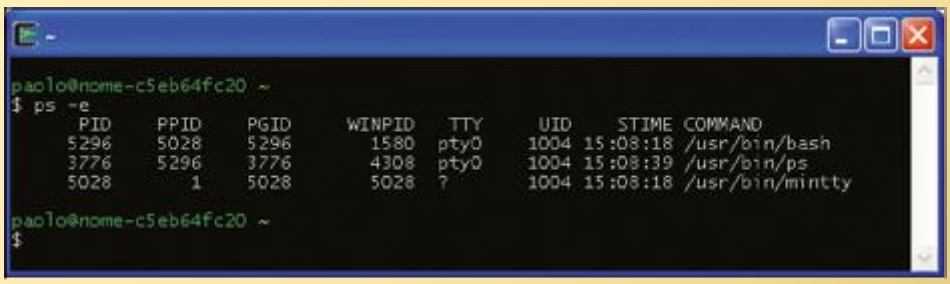
La sintassi dell'istruzione è la seguente:

```
int fork();
```

che per essere eseguita necessita dell'inclusione della seguente libreria:

```
#include <stdlib.h>
```

La funzione fork serve per **creare** un processo figlio identico al processo padre e tutti i segmenti del padre sono duplicati nel figlio al momento della fork.
All'atto della creazione del nuovo processo il sistema gli assegna un PID, come già descritto nella esercitazione 2, che viene restituito come parametro al processo padre mentre nel processo figlio la funzione ritorna 0; in caso di errore, cioè nel caso in cui non si fosse creato un nuovo processo, viene ritornato -1.

> Ogni processo figlio "si ricorda" il PID del processo padre che prende il nome di parent pid o ppid.
> È possibile visualizzare pid e ppid su tutti i processi che sono in esecuzione digitando da console il comando
>
> ```
> ps –e
> ```
>
> 
>

Per terminare l'esecuzione di un processo si richiama funzione exit:

```
void exit(int stato);
```

come parametro di ritorno ritorna un valore intero che però, per essere visualizzato correttamente, deve essere diviso per 256.

Vediamo un semplice esempio, evidenziando il codice che viene eseguito dai due processi.

Codice eseguito dal processo padre

```
#include <stdio.h>
#include <stdlib.h>
void main(){
 pid_t pid;
 pid=fork(); //
 if(pid==0){
   printf("1 processo figlio \n");
   printf("2 nel figlio pid= %d\n",pid);
   exit(1); // termina proc.figlio
 }else{
   printf("A processo padre \n");
   printf("B mio figlio ha pid=%d\n",pid);
 }
}
```

Codice eseguito dal processo figlio

```
#include <stdio.h>
#include <stdlib.h>
void main(){
 pid_t pid;
 pid=fork(); // qui pid = 0
 if (pid==0){
   printf("1 processo figlio \n");
   printf("2 nel figlio pid= %d\n",pid);
   exit(1); // termina proc.figlio
 }else{
   printf("A processo padre \n");
   printf("B mio figlio ha pid=%d\n",pid);
 }
}
```

Una possibile esecuzione è la seguente:

Prova adesso!

Scrivi un programma in linguaggio C che:

- genera due processi padre/figlio;
- il processo padre legge il nome e cognome dell'utente e lo trasforma con l'iniziale maiuscola e le altre lettere minuscole (cioè per esempio inserendo "rossi MARIO" produce "Rossi Mario");
- il processo padre scrive il nome così trasformato in un file di testo aperto prima di eseguire la fork;
- dopo aver completato la scrittura del nome, chiude e rimuove il file creato;
- contemporaneamente il processo figlio attende 5 secondi e stampa il contenuto del file appena scritto dal processo padre convertendo le parole in maiuscolo;

Dopo aver mandato in esecuzione più volte il programma:

- evidenzia le problematiche riscontrate;
- confronta la tua soluzione con l'esempio presente in fileMaiuMinu.C.

Suggerimento: per attendere 5 secondi utilizza la funzione "sleep (secondi)".

Wait

L'istruzione che attende la terminazione del processo figlio da parte del padre è la seguente

```
pid_t wait(int *retval);
```

che per essere eseguita necessita dell'inclusione nel programma delle seguenti librerie:

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
```

Come parametro di ritorno il processo chiamante riceve il valore del PID del processo figlio terminato mentre la variabile che viene passata (nel nostro esempio **retval**) sarà quella che conterrà il valore di terminazione impostato dalla funzione exit() del figlio, che viene letta mediante la macro

```
int WEXITSTATUS(retval)
```

Oltre alla funzione wait il linguaggio C mette a disposizione una istruzione più completa per attendere la terminazione di un processo, la funzione:

```
pid_t waitpid(pid_t pid, int* stato, int opzioni)
```

che ha tre parametri:

1. pid: serve a indicare quale processo si deve aspettare, e può essere:
   - < -1 attende della terminazione di un qualunque processo figlio il cui PGID (identificativo di ◀ process group ▶) è uguale al valore assoluto del parametro (per esempio, -5764 indica che il process group è 5764);
   - -1: attende la terminazione di un qualunque processo figlio, in modo analogo alla wait();
   - 0: attende un qualunque processo figlio il cui PGID è uguale a quello del processo chiamante;
   - > 0: attende la terminazione del processo figlio con uno specifico valore di PID.
2. status: permette di memorizzare il valore di ritorno del processo che si sta attendendo;
3. options: è utilizzato per specificare opzioni avanzate che esulano dai nostri scopi (per noi vale sempre 0).

◀ Process group Appartengono allo stesso process group, e quindi hanno PGID uguale, tutti i processi discendenti dallo stesso antenato padre oppure i processi raggruppati esplicitamente con la funzione setpgrp(). ▶

Due funzioni comode per realizzare programmi con molti figli sono:

```
pid_t getpid()
```

la quale ritorna il PID del processo che la esegue e

```
pid_t getppid()
```

che ritorna il PID del processo padre di chi la esegue.

ESEMPIO 15 *Programma orfano.c*

Come esempio che utilizza le funzioni sopra descritte scriviamo un programma dove il padre termina prima del figlio, lasciandolo "orfano": dopo che il padre avrà effettuato la exit(), la getppid() chiamata dal figlio ritornerà come valore 1, che è il ppid del processo init che lo ha "adottato".

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 int main(){
 5   int i;
 6   pid_t pid;
 7   pid = fork();
 8   if (pid > 0){
 9     printf("Padre si riposa\n");
10     sleep(3);
11     printf("Padre termina\n");
12     exit(0);
13   }
14   else if (pid == 0) {
15     while (getppid()>1){
16       printf("Mio padre e' %d\n", getppid());
17       sleep(1);
18     }
19     printf("Ora sono orfano!\n");
20   }
```

Una possibile esecuzione è la seguente: ▶

```
paolo@nome-c5eb64fc20 ~
$ ./orfano
Mio padre e' 3096
Padre si riposa
Mio padre e' 3096
Mio padre e' 3096
Padre termina
Mio padre e' 3096

paolo@nome-c5eb64fc20 ~
$ Ora sono orfano!
```

Prova adesso!

Scrivi un programma che produca un risultato simile a quello sotto riportato, dove un processo padre genera quattro figli che vivono 2 secondi e poi terminano la loro esecuzione.

```
paolo@nome-c5eb64fc20 ~
$ ./multiFork
Figlio (pid=7020, ppid=6236) in sleep per 2 secondi
Figlio (pid=2716, ppid=6236) in sleep per 2 secondi
Figlio (pid=7696, ppid=6236) in sleep per 2 secondi
Figlio (pid=608, ppid=6236) in sleep per 2 secondi
Padre (pid=6236, ppid=5768) riceve da figlio 7020 exit status 1
Padre (pid=6236, ppid=5768) riceve da figlio 2716 exit status 1
Padre (pid=6236, ppid=5768) riceve da figlio 7696 exit status 1
Padre (pid=6236, ppid=5768) riceve da figlio 608 exit status 1

paolo@nome-c5eb64fc20 ~
$
```

Il padre attende la terminazione dei figli e ne legge lo stato di uscita.
Confronta la tua soluzione con quella riportata nel file multiFork.c

Espressione matematica

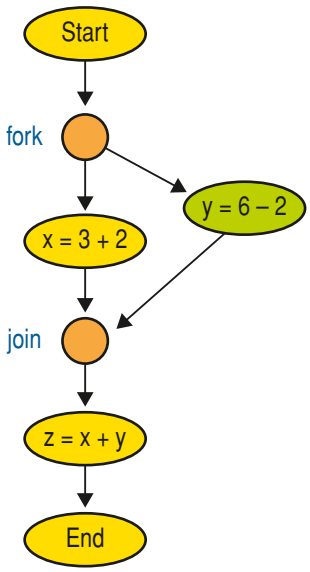
Scriviamo ora un programma che esegue un calcolo parallelo dove il processo figlio passa il risultato al processo padre per generare il risultato finale.

Data la seguente espressione matematica:

$$z = (3+2) * (4-6)$$

scriviamo un programma che la esegue col massimo parallelismo.

Per prima cosa realizziamo il grafo delle precedenze parallelizzando le operazioni indicate tra parentesi:



In pseudocodifica viene codificato in:

```
/* processo padre: */
start
   p2 = fork figlio1;  // inizia l'elaborazione parallela
   x=3+2;
   join p2;            // termina l'elaborazione parallela
   z=x+y;
…
end.

/* codice nuovo processo:*/
int figlio1()
{
  y=6-2;
  return y
}
```

Prova adesso!

Codifica il programma in linguaggio C.
Confronta la tua esecuzione con quella riportata nella figura seguente:

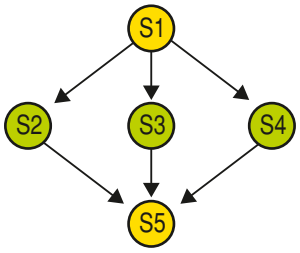
```
paolo@nome-c5eb64fc20 ~
$ gcc forkjoin2.c -o forkjoin2

paolo@nome-c5eb64fc20 ~
$ ./forkjoin2
elab. parallela processo figlio
elab. parallela processo padre
join: padre aspetta
elab. finale padre
risultato finale z= 9
paolo@nome-c5eb64fc20 ~
$
```

Confronta il tuo codice con quello presente nel file forkjoin2.c.

Cobegin-coend

Mentre il costrutto fork-join permette di realizzare qualsiasi grafo di precedenza fra task, il costrutto cobegin-coend è più restrittivo e a volte non permette di descrivere tutte le situazioni di concorrenza. Il linguaggio C non ha una istruzione specifica per questo costrutto, ma è semplicemente realizzabile mediante le istruzioni fork() e wait() appena descritte.

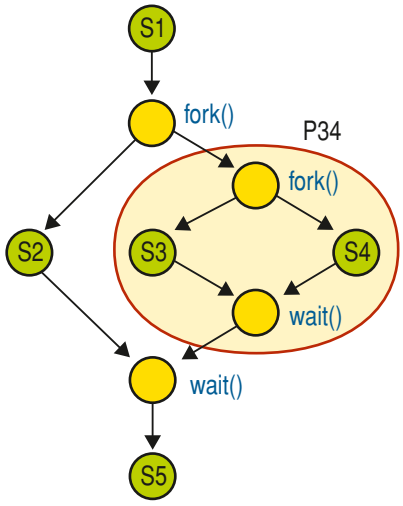
L'esempio di figura prevede l'esecuzione parallela di tre processi ▶



descritta con la seguente pseudocodifica

```
S1;
cobegin
S2;
S3;
S4;
coend
S5;
```

Può essere vista come la combinazione di più fork-wait() come riportata nel seguente diagramma



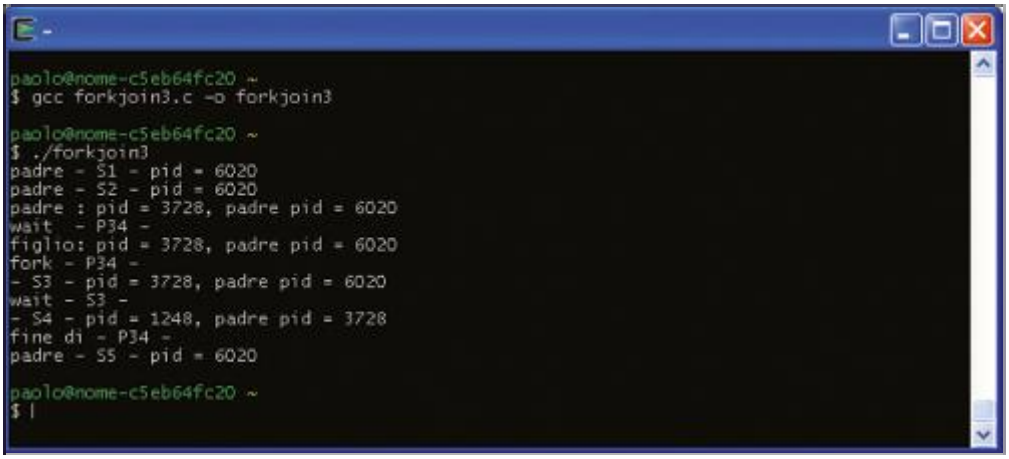
La codifica in linguaggio C del primo segmento che esegue la prima fork è il seguente:

```
11 int main( ){
12   pid_t childpid;
13   printf ("padre - S1 - pid = %d\n", getpid());
14   if ((childpid = fork()) == -1 )
15    printf( "fork non riuscita !" );
16   if (childpid == 0){
17    printf("figlio: pid = %d, padre pid = %d\n", getpid(), getppid());
18    ramo34();
19   }else {
20    printf("padre - S2 - pid = %d\n",getpid());
21    printf("padre : pid = %d, padre pid = %d\n", childpid, getpid());
22    printf("wait  - P34 - \n" );
23    while (wait( (int *)0 ) != childpid );  // aspetta terminazione figlio
24    printf("padre - S5 - pid = %d\n",getpid());
25    exit( 0 );
26   }
27 }
```

La codifica del secondo segmento, quello che esegue la seconda fork, è il seguente:

```
28
29 ramo34(){
30   pid_t pid4;
31   printf( "fork - P34 - \n" );
32   if (( pid4 = fork() ) == -1 )
33     printf( "fork non riuscita !" );
34   if (pid4 == 0 ){
35     printf("- S4 - pid = %d, padre pid = %d\n", getpid(), getppid() );
36   }else{
37     printf("- S3 - pid = %d, padre pid = %d\n", getpid(), getppid() );
38     printf("wait - S3 - \n" );
39     while (wait( (int *)0 ) != pid4 ); // aspetta terminazione figlio
40     printf("fine di - P34 -\n" );
41   }
42  exit( 0 );
43 }
```

Mandandolo in esecuzione, un possibile output è il seguente:



Fork annidate

Scrivi, compila e manda in esecuzione il seguente programma:

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
  int pid1=fork();
  if (pid1==0) {
    printf("Sono un figlio, il mio pid e': %d. ", getpid());
    printf("Il mio papi ha pid: %d\n", getppid());
    exit(1) ;      /* termina processo figlio */
  ) else {
    printf("Sono un padre: il mio pid e': %d.\n", getpid());
    exit(0) ;      /* non necessario */
  )
}
```

Otterrai la seguente videata:

Ora modifica il codice come riportato nella seguente figura:

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <sys/wait.h>
4 #include <fcntl.h>
5 #include <stdio.h>
6 #include <unistd.h>
7 #include <stdlib.h>
8
9 int main() {
10   int pid1=fork();
11   int pid2=fork();
12   int pid3=fork();
13   if ((pid1==0)||(pid2==0)||(pid3==0)) {
14     printf("Sono un figlio, il mio pid e': %d. ", getpid());
15     printf("Il mio papi ha pid: %d\n", getppid());
16     exit(1) ;      /* termina processo figlio */
17   } else {
18     printf("Sono un padre: il mio pid e': %d.\n", getpid());
```

Prova adesso!

Prima di mandare in esecuzione il programma, rispondi alle seguenti domande:

- Cosa ti aspetti di vedere sullo schermo?
- Quanti figli vengono generati?
- Perché?

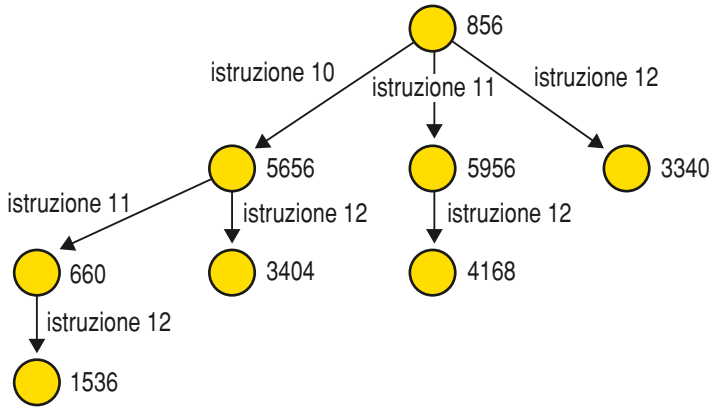
Soluzione

Mandando in esecuzione il programma si ottiene un output simile a quello riportato nella figura seguente:

```
paolo@nome-c5eb64fc20 ~
$ ./fork2
Sono un figlio, il mio pid e': 1536. Il mio papi ha pid: 660
Sono un figlio, il mio pid e': 660. Il mio papi ha pid: 5656
Sono un figlio, il mio pid e': 4168. Il mio papi ha pid: 5956
Sono un figlio, il mio pid e': 5956. Il mio papi ha pid: 856
Sono un figlio, il mio pid e': 3404. Il mio papi ha pid: 5656
Sono un figlio, il mio pid e': 5656. Il mio papi ha pid: 856
Sono un padre: il mio pid e': 856.
Sono un figlio, il mio pid e': 3340. Il mio papi ha pid: 856

paolo@nome-c5eb64fc20 ~
$
```

Vengono quindi generati sette processi figli (e non tre come probabilmente hai erroneamente risposto).

Per comprendere la dinamica del programma disegniamo graficamente l'elenco delle fork e assegniamo a ogni processo il suo pid:



La prima fork (istruzione 10) genera un processo che a sua volta esegue la seconda fork (istruzione 11), quindi i primi due processi generano altri due figli che a loro volta eseguiranno la fork della istruzione 12: in totale sono (padre+ figlio) + 2 processi + 4 processi per un totale di otto processi (2^3.processi). Allo stesso modo avendo 10 fork di seguito genereranno 2^{10}.

Esecuzione non deterministica

L'esecuzione di un programma parallelo viene influenzata dallo scheduler del sistema operativo e quindi l'out sullo schermo può essere diverso per ogni sua esecuzione.
Per verificarlo scriviamo un semplice programma.

Prova adesso!

Scrivi un programma concorrente dove un processo padre genera solamente due processi figli. Mandalo in esecuzione più volte confrontando il risultato.

Una possibile soluzione è la seguente:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void main(){
  pid_t pid,miopid;
  pid=fork();
  if(pid==0){
    miopid=getpid();
    printf("1)sono il primo processo figlio con pid: %i\n",miopid);
    exit(1);        /* termina primo processo figlio */
  }else{
    printf("2)sono il processo padre\n");
    printf("3)ho creato un processo con pid: %i\n", pid);
    miopid=getpid();
    printf("4)il mio pid e' invece: %i\n", miopid);
    pid=fork( );
    if (pid==0){
      miopid=getpid();
      printf("5)sono il secondo processo figlio con pid: %i\n",miopid);
      exit(2);      /* termina secondo processo figlio */
    }else {
      printf("6)sono il processo padre\n");
      printf("7)ho creato un secondo processo con pid: %i\n", pid);
      exit(0);      /* non necessario */
    }
  }
}
```

Una prima esecuzione dà il seguente output:

Una seconda esecuzione dà il seguente output:

Zoom su...

Per intercettare l'esecuzione dei due processi si può anche utilizzare una istruzione di switch prendendo il pid come selettore, come nell'esempio seguente:

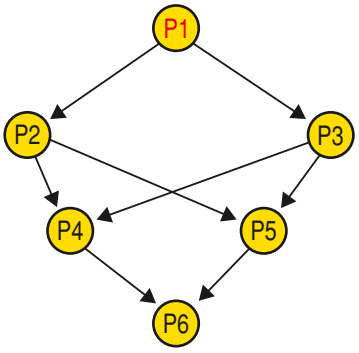
```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
 int pid,retv;
 pid=fork();
 switch(pid){
  case -1: printf("Errore in creazione figlio\n");
    return(-1);
  case 0 :                  /* figlio */
    printf("Figlio si riposa  .. \n");
    sleep(5);
    printf("Figlio termina\n");
    exit(69);               /* termina il processo figlio */
  default:
    printf("Padre aspetta che termina il figlio...\n");
    wait(&retv);            /* figlio aspetta che termina il padre */
    printf("Il figlio ha terminato con exit status %d\n",WEXITSTATUS(retv));
    printf("Padre termina \n");
    exit(1);                /* termina il processo padre */
 }
}
```

Una esecuzione dà il seguente output:

```
paolo@nome-c5eb64fc20 ~
$ gcc fork6.c -o fork6

paolo@nome-c5eb64fc20 ~
$ ./fork6
Figlio si riposa ..
Padre aspetta che termina il figlio...
Figlio termina
Il figlio ha terminato con exit status 69
Padre termina

paolo@nome-c5eb64fc20 ~
$
```

Esercizi di approfondimento

1 Partendo dal grafo delle precedenze di figura scrivi il programma concorrente in pseudolinguaggio utilizzando il costrutto fork-join. Quindi codificalo in linguaggio C.



Le istruzioni che devono essere eseguite sono le seguenti:

```
/* codice P1 */
leggi (x,y)
a:= x + y;

/* codice P2 */
leggi(z)
b:= z + a;

/* codice P3 */
c:= a - b;

/* codice P4 */
w:= (b + c)*2;

/* codice P5 */
d:= (b - c)*2;

/* codice P6 */
f:= w * d;
```

2 Realizza un programma concorrente che calcoli le prime N potenze di un dato X?

Esercitazioni di Laboratorio 4

I THREAD IN C

Info

I codici sorgenti sono nel file C_pthread.rar scaricabile dalla cartella materiali nella sezione del sito www.hoepliscuola.it riservata al presente volume.

L'implementazione dei thread in LINUX

Per utilizzare i pthread in un programma C è necessario includere la libreria:

```
<pthread.h>
```

In questo modo potremmo utilizzare i thread definiti dallo standard POSIX, Portable Operating System Interface for Computing Environments, che prendono proprio il nome di "pthread" e sono stati definiti con l'obiettivo di avere uno standard di riferimento che permetta la portabilità dei programmi in ambienti diversi che utilizzano le interfacce applicative (API) dei sistemi operativi.

Linux non è completamente POSIX compatibile per quanto riguarda la gestione dei segnali, e dalla versione RedHat si è sviluppato il supporto nativo ai thread per Linux (NPTL: Native POSIX Thread Linux) che oltre avere vantaggi in termini di prestazioni è maggiormente aderente allo standard POSIX.

Creazione di thread: pthread_create

La chiamata per creare un thread è definita da questo prototipo:

```
pthread_create(pthread_t* ptID, pthread_attribute attr, void* routine, void* arg)
```

Dove:

- ptID: è il puntatore alla variabile che contiene l'identificativo del thread che viene creato, il thread_ID (tID);
- arg: è il puntatore all'eventuale vettore contenente i parametri da passare al thread creato e può essere posto a NULL nel caso di assenza di parametri;
- routine: è il puntatore alla funzione che contiene il codice che il thread deve eseguire;
- att: è il puntatore all'eventuale vettore contenente i parametri da passare al thread creato (per esempio la priorità); noi utilizzeremo sempre gli attributi di default e quindi indicheremo nella chiamata NULL.

La funzione restituisce 0 in caso che vada a buon fine la creazione di un nuovo thread, altrimenti ritorna un codice di errore diverso da zero.

Vediamo un primo esempio di creazione ▶

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 void* codice_thread (void* arg){
6   printf ("il pid del figlio e' %d\n", (int) getpid ());
7   return NULL;
8 }
9
10 int main (){
11   pthread_t miothread;
12   printf ("il pid del main is %d\n", (int) getpid ());
13   pthread_create (&miothread, NULL, &codice_thread, NULL);
14   return 0;
15 }
```

Che, mandato in esecuzione, visualizza una situazione simile alla seguente: ▶

```
~/pthread
paolo@nome-c5eb64fc20 ~/pthread
$ gcc pThreadErr.c -o pThreadErr

paolo@nome-c5eb64fc20 ~/pthread
$ ./pThreadErr
il pid del main is 3456

paolo@nome-c5eb64fc20 ~/pthread
$
```

Non visualizzando l'output della riga 6 possiamo ipotizzare che il codice del thread non venga eseguito: molto probabilmente il processo padre termina prima che avvenga la schedulazione del thread e quindi, alla sua terminazione, comporta la terminazione anche del figlio "prima che nasca".

È quindi necessario introdurre delle istruzioni di ritardo nel padre affinché possa permettere la nascita del figlio stesso, adesso modificando il codice, come di seguito riportato, inserendo un secondo di ritardo nel processo padre. ▶

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 void* codice_thread (void* arg){
6   printf ("il pid del figlio e' %d\n", (int) getpid ());
7   return NULL;
8 }
9
10 int main (){
11   pthread_t miothread;
12   printf ("il pid del main e' %d\n", (int) getpid ());
13   pthread_create (&miothread, NULL, &codice_thread, NULL);
14   sleep(1);
15   return 0;
16 }
```

che produrrà il seguente output: ▶

dove possiamo riconoscere oltre al padre anche l'esecuzione del thread figlio.

```
~/pthread
paolo@nome-c5eb64fc20 ~/pthread
$ gcc pThreadA.c -o pThreadA

paolo@nome-c5eb64fc20 ~/pthread
$ ./pThreadA
il pid del main is 6084
il pid del figlio e' 6084

paolo@nome-c5eb64fc20 ~/pthread
$
```

Il pid visualizzato è sempre quello del processo padre, dato che il thread vive all'interno del padre.

È sempre meglio verificare il buon fine della creazione del thread introducendo il controllo sul parametro di ritorno della creazione del thread.

Prova adesso!

Scrivi un programma che generi un thread che esegue il conto alla rovescia a partire da 10 fino ad arrivare a 0 visualizzandolo sullo schermo.
Il processo padre si sospende in attesa della terminazione del thread: inserisci il controllo sul buon esito della creazione del thread.

Soluzione

Una possibile soluzione è la seguente

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <pthread.h>
 4
 5 void *codice_thread(void *arg){
 6   int k;
 7   for (k=10; k>0; k-- ){
 8    printf("Il thread esegue il countdown : -%d\n",k);
 9    sleep(1);
10   }
11   printf("Svegliaaaaaaaaaaaaaaaaaa !!! \n\n");
12   return NULL;
13 }
14
15 int main(void){
16   pthread_t miothread;
17   if (pthread_create(&miothread,NULL,codice_thread,NULL)){
18    printf("errore nella creazione del thread."); abort();
19   }
20   printf(" Ho creato un thread ... ora dormo e lo aspetto  \n\n");
21   sleep(15);
22   printf(" Mi risveglio e termino l'elaborazione \n\n");
23   exit(0);
24 }
```

e una sua esecuzione produrrà il seguente output:

```
~/pthread
paolo@nome-c5eb64fc20 ~/pthread
$ gcc pThread3.c -o pThread3

paolo@nome-c5eb64fc20 ~/pthread
$ ./pThread3
Il thread esegue il countdown : -10
Il thread esegue il countdown : -9
Il thread esegue il countdown : -8
Il thread esegue il countdown : -7
Il thread esegue il countdown : -6
Il thread esegue il countdown : -5
Il thread esegue il countdown : -4
Il thread esegue il countdown : -3
Il thread esegue il countdown : -2
Il thread esegue il countdown : -1
Svegliaaaaaaaaaaaaaaaaa !!!

 Mi risveglio e termino l'elaborazione
```

Terminazione di thread: pthread_exit

Esistono due modalità per terminare l'esecuzione di un thread:

A quella tradizionale che avviene alla fine dell'esecuzione del codice della funzione, cioè la classica

```
return();
```

B utilizzando la chiamata alla funzione

```
void pthread_exit(void *retval);
```

con retval che è il puntatore alla variabile che contiene il valore di ritorno e, come vedremo, potrà essere letto dal processo padre con la funzione join.

Prova adesso!

Scrivi un programma che generi due thread: il primo scrive una frase di saluto sullo schermo e il secondo scrive il tuo nome e cognome definite in stringhe globali.
Utilizza la funzione pthread_exit per terminare l'elaborazione di entrambi i thread.

Puoi confrontare la tua soluzione con quella presente nel file pThread4.c

Modifica ora il programma precedente in modo che i due thread eseguano lo stesso segmento di codice ma prendano come parametro la stringa da visualizzare.

L'esempio seguente mostra come viene passato un parametro a un thread.

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 void* funzione (void* parametro){
6   char* messaggio;
7   messaggio= (char *)parametro;
8   printf ("%s \n", messaggio);
9   pthread_exit(0);
10 }
11
12 int main (){
13   char *msg1 =" Ciao: sono una stringa passata dal processo padre";
14   pthread_t mioThread;
15   pthread_create (&mioThread, NULL, &funzione,(void*) msg1 );
16   sleep(1);
17
18   return 0;
19 }
```

Puoi confrontare la tua soluzione con quella presente nel file pThread6.c

Attesa di thread: pthread_ join

La modalità corretta per attendere la terminazione di un thread (invece che introdurre una istruzione di sleep(tempo) con un valore del tempo impostato "a caso") è quella di utilizzare la funzione

```
int pthread_join(pthread_t mioThread, void **par_ritorno);
```

Dove:

- mioThread: è il pid del thread del quale si vuole attendere la terminazione;
- par_ritorno: in questa variabile viene memorizzato il valore di ritorno del thread che il thread terminato ha passato a pthread_exit(parametro): può quindi avere valore NULL in caso di assenza di parametro di ritorno.

> È possibile mettere un unico processo/thread in attesa della terminazione di un altro thread.

Il seguente esempio illustra come utilizzare questa funzione: il processo padre passa al thread come parametro un numero intero che questo, dopo averlo raddoppiato, ritorna al chiamante.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  void* codice(void* parametroIN){
6    int parametroOUT;
7    int valore1=(int)parametroIN;
8    parametroOUT= valore1*2;
9    pthread_exit((void*)parametroOUT);
10 }
11 int main () {
12   pthread_t t1;
13   int risultato, dato;
14   dato=10;
15   pthread_create(&t1,NULL,codice,(void*) dato);
16   pthread_join(t1,(void*)&risultato);
17   printf("risultato %d\n",risultato);
18   return 0;
19 }
```

Prova adesso!

Scrivi un programma che gestisca due variabili globali di tipo intero (per esempio int numero1=200, numero2=300;) e chiama due thread per "spostare" parte del primo numero nel secondo numero.
La quantità da "spostare" viene passata come parametro alla creazione dei due thread.
Il processo principale attende la terminazione dei due figli e visualizza i nuovi valori delle due variabili e il loro totale.

Puoi confrontare la tua soluzione con quella presente nel file pThread8.c

Esercitazioni di Laboratorio 5

I THREADS IN JAVA: CONCETTI BASE

Info

I codici sorgenti sono nel file java_THREAD.rar scaricabile dalla cartella materiali nella sezione del sito www.hoepliscuola.it riservata al presente volume.

I threads in Java

Il linguaggio Java già dalla sua prima definizione mette a disposizione dell'utente classi che implementano i thread in forma primitiva ed è anche questo un motivo per cui viene utilizzato come linguaggio di progetto nei corsi per sistemi operativi.

Esistono tre possibilità per definire un thread in Java:
1. creare un main();
2. creare una sottoclasse della classe Thread;
3. implementare l'interfaccia Runnable.

La prima modalità di creazione di un thread è implicita nella realizzazione di un programma Java in quanto ogni programma Java contiene almeno un singolo thread, corrispondente all'esecuzione del metodo main() sulla JVM.
La seconda modalità consiste nel creare i Thread come oggetti di sottoclassi della classe Thread.
La terza modalità consiste nell'utilizzo dell'interfaccia Runnable che risulta avere maggior flessibilità del caso precedente.

In questa esercitazione vedremo come scrivere e come generare un thread dinamicamente attivando concorrentemente la sua esecuzione all'interno del programma.

Dato che in un'applicazione si può avere un unico main()non sarà possibile utilizzare tale metodo per creare applicazioni concorrenti: dobbiamo comunque tenere sempre presente che l'esecuzione di un main implica la creazione di un thread che verrà schedulato insieme agli altri thread che definiremo in seguito.

Thread come oggetti di sottoclassi della classe Thread

Java mette a disposizione la classe Thread (fornita dal package **java.lang**), fondamentale per la gestione dei thread, in quanto in essa sono definiti i metodi per avviare, fermare, sospendere, riattivare e così via, come si vede dal diagramma della classe.

Il diagramma delle classi è il seguente:

Classe Thread		
Attributi	Metodi costruttori	Metodi modificatori
int MIN_PRIORITY, NORM_PRIORITY, MAX_PRIORITY	Thread() Thread(Runnable target) Thread(String name) Thread(ThreadGroup name, Runnable target) Thread(ThreadGroup group, String name) Thread(Runnable target, String name) Thread(ThreadGroup group, Runnable target, String name) Thread(ThreadGroup group, Runnable target, String name, long stackSize)	int activeCount() Thread currentThread() void dumpStack() int enumerate(Thread tarray[]) boolean holdsLock(Object obj[]) boolean interrupted() void sleep(long millis) void sleep(long millis, int nanos) void yield() --- ClassLoader get/setContextClassLoader() String get/setName() int get/setPriority() ThreadGroup getThreadGroup() boolean isAlive() boolean is/setDaemon() boolean isInterrupted() --- String toString() --- void checkAccess() void destroy() void interrupt() void join() void join(long millis) void join(long millis, int nanos) void run() void start()

La prima modalità di creazione di thread in Java viene realizzata ereditando da tale classe una sottoclasse che utilizzi il metodo run() con la seguente segnatura:

```
public void run(void)
```

Il metodo run definisce l'insieme di istruzioni Java che ogni thread "creato come oggetto della classe" Thread eseguirà concorrentemente con gli altri thread.

> Nella classe Thread l'implementazione del suo metodo run è vuota, quindi in ogni sottoclasse derivata deve essere ridefinito (override) con le istruzioni che costituiscono il corpo del thread, cioè tutte le istruzioni che lo schedulatore deve eseguire quando il thread viene attivato.

ESEMPIO 16 *Il thread dei 7 nani*

Un esempio completo è il seguente: ▶

```
public class ContaINanil extends Thread{
  public void run(){
    setName("settenani");
    System.out.println(Thread.currentThread().getName());
    for(int i=0;i<7;i++){
      System.out.print((i+1)+" ");}
  }
}
```

Questa semplice classe definisce un segmento di codice che, dopo aver assegnato il nome "settenani" al thread con il metodo setName() e averlo visualizzato sullo schermo, conta da 1 a 7.
Per l'esecuzione di un thread che esegua questo codice si deve scrivere una classe di prova che ne definisca un'istanza e ne avvii l'esecuzione: deve essere chiamato il metodo start() che invoca il metodo run() (il metodo run() non può essere chiamato direttamente, ma solo attraverso start()).

```
public class ProvaNanil{
  public static void main(String args[]){
    ContaINanil thr1 = new ContaINanil();
    thr1.start();
    // o in una unica istruzione : new ContaINani().start();
    System.out.println(Thread.currentThread().getName());
  }
}
```

Effettuiamo cioè la definizione di un oggetto in una classe di prova e attiviamo il thread con il metodo start() che manda in esecuzione il metodo run() prima definito.

> La creazione di un oggetto thread NON determina l'esecuzione: il thread è ancora in uno stato simile allo stato di attesa, cioè è in attesa di essere avviato!

In questo modo abbiamo creato due thread concorrenti:
- il thread principale, associato al main;
- il thread thr1 creato dinamicamente dal precedente, con l'esecuzione dell'istruzione thr1.start() che lancia in concorrenza l'esecuzione del metodo run() del nuovo thread.

Una esecuzione produce sullo schermo il seguente output:

> A ogni thread viene associato automaticamente un nome: il primo thread ha nome main ed è inconfondibile dato che deve essere unico; ai successivi Thread viene assegnato di default un nome con un indice progressivo (per esempio Thread-6)a meno che il programmatore provveda alla sua inizializzazione esplicitamente, come fatto da noi.

Il thread appena creato continua la sua evoluzione fin tanto che viene completata l'esecuzione del codice, ovvero finché il contatore ha raggiunto e visualizzato il numero 7; quindi termina spontaneamente.
In altri casi, potremmo avere necessità di fermare espressamente l'evoluzione del thread tramite un altro thread: tale meccanismo viene realizzato semplicemente con l'invocazione del metodo stop()

```
thr1.stop();
```

ESEMPIO 17 *Due thread in esecuzione*

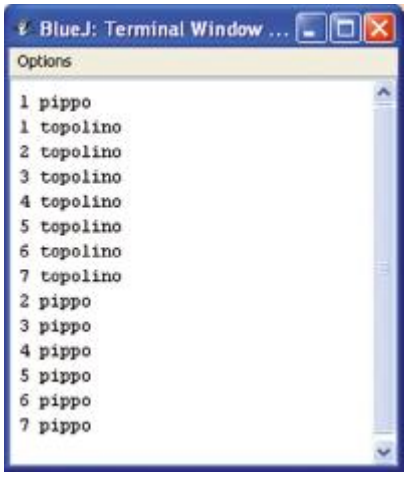
Vediamo un secondo esempio dove inseriamo un costruttore per assegnare il nome al thread:

```
public class ContaINani2 extends Thread{
  public ContaINani2(String nome){          // costruttore
    super();
    setName(nome);
  }
  public void run(){
    for(int i=0;i<7;i++){
      System.out.println((i+1)+" "+getName());}
  }
}
```

Modifichiamo ora la classe di prova in modo che vengano create le due istanze con i nomi diversi:

```
public class ProvaNani2{
  public static void main(String args[]){
    Thread thr1 = new ContaINani2("topolino");
    Thread thr2 = new ContaINani2("pippo");
    thr1.start();
    thr2.start();
  }
}
```

Due esecuzioni producono sullo schermo il seguente output:



Si può osservare come gli effetti della schedulazione producano risultati diversi.

Prova adesso!

1. Modifica la classe di prova in modo che vengano create le sette istanze con i nomi diversi, per esempio quelli dei sette nani.
2. Avvia l'esecuzione e osserva e commenta l'output.

Thread come classe che implementa l'interfaccia Runnable

La terza possibilità consiste nell'utilizzo dell'interfaccia Runnable: con questo meccanismo si ha maggiore flessibilità rispetto a quello appena descritto.

La creazione della classe è simile al metodo precedente e le operazioni da eseguire sono le seguenti:

- codificare il metodo run() nella classe che implementa l'interfaccia Runnable;
- creare un'istanza della classe tramite new;
- creare un'istanza della classe Thread con un'altra new, passando come parametro l'istanza della classe che si è creata;
- invocare il metodo start() sul thread creato, producendo la chiamata al suo metodo run().

ESEMPIO 18 *Campane non sincronizzate*

Scriviamo un esempio classico, le "campane che suonano".
Costruiamo la classe Campana e come parametri del costruttore indichiamo il suono che deve emettere e quante volte dovrà essere eseguito.

```
class Campana implements Runnable{
  String suono;
  int volte;
  public Campana(String suono,int volte){
    this.suono =suono;
    this.volte=volte;
  }
  public void run(){
    for(int i=0;i<volte;i++) {
      System.out.print ((i+1)+suono+" ");}
  }
}
```

Quindi, definiamo la classe che crea tre thread e li manda in esecuzione:

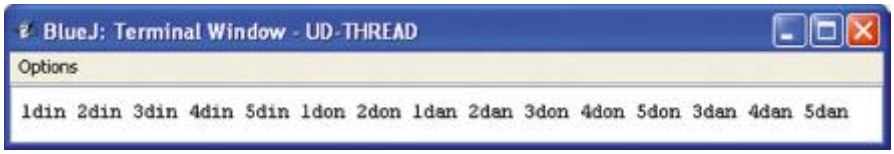
```
public class Dindondan{
  public static void main(String args[]){
    //prima modalità di definizione
    Runnable cam1 = new Campana("din",5);
    Thread thr1 = new Thread(cam1);
    thr1.start();
    //seconda modalità di definizione
    Thread thr2 = new Thread(new Campana("don",5));
    thr2.start();
    // terza modalità di definizione
    new Thread(new Campana("dan",5)).start();
  }
}
```

Facciamo una prima osservazione sulla differenza del codice dei metodi run(): in questo caso non si è potuto utilizzare il metodo getName() per stampare il nome del thread (e usare tale nome come suono delle campane): infatti, gli oggetti della classe Campana non appartengono alla classe Thread ma implementano solamente l'interfaccia Runnable che ha un unico metodo run().
Si sono dovuti creare successivamente nel main() oggetti di tipo Thread passandogli come parametro del costruttore l'oggetto Runnable creato in precedenza.

Una caratteristica comune del metodo run() è invece l'assenza di parametri, sia in ingresso sia in uscita. Questo comporta che, qualora ci sia la necessità di passare parametri al thread, dobbiamo servirci del costruttore, passare tali dati come parametri e scriverli nelle variabili di istanza della classe.
Il metodo run(), così come tutti i metodi della classe, ha visibilità di tali variabili di classe e quindi è possibile accedervi, leggerle e modificarle.

Mandiamo ora in esecuzione la classe Dindondan e otteniamo il seguente output:

mentre una successiva esecuzione dà il seguente risultato, diverso dal precedente:



Quindi, l'output non è sempre uguale bensì dipende dalla schedulazione e dai processi/thread che al momento sono nella lista dei processi pronti: questa modalità è quindi non deterministica ed è una delle cause che rendono complicata la gestione della concorrenza.
Se abbiamo bisogno della perfetta "rotazione" dei suoni delle campane, e cioè din seguito da don, seguito da dan, dobbiamo implementare i meccanismi di sincronizzazione e cooperazione che vedremo in seguito.

Anche se può sembrare più complessa, questa seconda modalità è preferibile per realizzare i thread; infatti generalmente le classi che stiamo definendo sono ereditate da altre e, poiché Java non permette l'ereditarietà multipla, per avere anche le caratteristiche dei thread l'unica soluzione è quella di usare implements (come già visto per la gestione degli eventi).

Prova adesso!

1. Modifica il programma precedente cercando di ottenere sullo schermo una sequenza ben determinata di suoni, per esempio tre serie di "din-don-dan" semplicemente introducendo dei ritardi per esempio con il metodo sleep(secondi), che blocca per un tempo specificato l'esecuzione di un thread.
2. Manda in esecuzione più volte il programma per essere sicuro di aver ben sincronizzato (?!) il sistema: quali osservazioni puoi fare?
3. Modifica ora il programma introducendo una nuova campana dal suono "dun", che deve essere alternata a ciascuna altra campana, cioè deve sempre seguire una delle tre campane precedenti.
   Ad esempio una sequenza possibile è la seguente:
   "din-dun" - don-dun - din-dun - dan-dun"

Rilascio del processore

Sino a ora abbiamo visto tre possibili situazioni in cui il thread lascia il processore:
- alla terminazione dell'esecuzione del codice presente nel metodo run(), e in questo caso è un rilascio "definitivo";
- quando lo schedulatore lo mette nello stato di pronto al termine del suo quanto di tempo;
- quando il processo necessita di una risorsa non disponibile e quindi viene messo nello stato di attesa.

La classe Thread offre inoltre un insieme di metodi, che chiamiamo *di controllo*, i quali ci permettono di gestire l'uso del processore o, meglio, di liberare l'utilizzo del processore:
- stop(): viene invocato da un altro oggetto che lo termina definitivamente;
- suspend(): viene invocato da un altro oggetto che lo sospende temporaneamente;
- resume(): viene invocato da un altro oggetto che lo riattiva dalla sospensione;
- sleep(int): viene invocato da un oggetto che si sospende per int di millisecondi.

I primi tre metodi sono sconsigliabili perché non sono sicuri, in quanto agiscono in modo diretto su un thread senza alcun controllo del suo stato o della sua evoluzione; possono quindi determinare situazioni di stallo e di *deadlock* (lezione 7 dell'unità di apprendimento 2):
- se fermiamo il thread appena questi ha acquisito il controllo di una risorsa, quest'ultima rimane a lui allocata e quindi indisponibile a tutto il sistema;
- se successivamente un thread ha necessità di utilizzare tale risorsa rimarrebbe "perennemente" in attesa;
- se fermiamo il thread mentre sta effettuando un'operazione critica e non la termina, ci portiamo in una situazione di inconsistenza.

Nessun problema per il metodo sleep(int), in quanto sospende solo temporaneamente il thread e lo riattiva dopo int millisecondi ripartendo dalla stessa istruzione dalla quale è stato interrotto.
Esiste un meccanismo che non interrompe immediatamente il thread ma gli segnala la necessità che si sospenda, lasciandogli però la scelta del momento opportuno, che sarà un momento non critico: tale meccanismo viene realizzato mediante il metodo yield()che consente di trasferire il controllo del processore a un altro thread (cede il controllo all'altro thread, rimettendosi in coda, in attesa di poter riacquisire l'esecuzione). Questa situazione prende il nome di coroutining.

ESEMPIO 19 *Rilascio col metodo yield()*

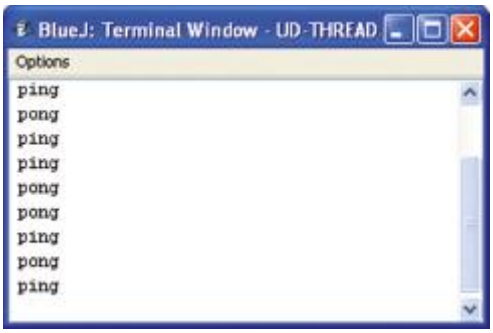
Vediamo un esempio che simula una partita di ping pong iniziando con la definizione di una classe Racchetta che nel metodo run() scrive il contenuto della variabile pallina, si sospende per un secondo (per facilitare la visualizzazione sullo schermo) e quindi cede l'esecuzione a un altro thread con il metodo yield().

```
class Racchetta implements Runnable{
  String pallina;
  public Racchetta(String pallina){
    this.pallina =pallina;
  }
  public void run(){
    while (true){
      System.out.println(pallina);
        try{
          Thread.sleep(1000);}  // ritardo solo per visualizzazione
        catch (InterruptedException e) {}
        Thread.yield();         // cede l'esecuzione ad un altro thread
    }
  }
}
```

La classe che crea i due thread è la seguente:

```
public class PingPong{
  public static void main(String args[]){
    Thread thr1 = new Thread(new Racchetta("ping"));
    thr1.start();
    // secondo giocatore
    Thread thr2 = new Thread(new Racchetta("pong"));
    thr2.start();
  }
}
```

L'eco sullo schermo è una sequenza infinita di scritte "ping pong" (a ogni secondo).



Nel metodo run() della classe Racchetta è stato invocato il metodo sleep() nel seguente modo: Thread.sleep(1000);.
Ciò è possibile per due motivi:

- sleep() è un metodo statico (polimorfo): fa sì che il thread che lo invoca venga posto in stato di attesa e reinserito nella lista dei processi pronti allo scadere dei millisecondi (o millisecondi + nanosecondi) specificati mediante i parametri;
- la classe Thread è totalmente integrata nel linguaggio Java e non inclusa in una libreria aggiuntiva.

L'utilizzo del metodo sleep() deve però essere effettuato implementando un meccanismo try/catch, in quanto può lanciare eccezioni InterruptedException, le quali quindi devono essere catturate e gestite.

Prova adesso!

1. Realizza un programma che simuli la corsa alla merenda dei tre nipoti di zio Paperino facendo avanzare casualmente la posizione di ciascuno di essi e verificando se viene raggiunto il traguardo: il primo che raggiunge i 10 m mangia la merenda.
2. Utilizza per la creazione dei thread entrambi i meccanismi descritti in questa esercitazione.
3. Verifica la tua soluzione con quella presente nella classe CorsaAllaMerenda.

Esercitazioni di Laboratorio

UN ESEMPIO CON I JAVA THREAD: CORSA DI BICICLETTE

Un esempio completo: thread e grafica

Si vuole realizzare un programma che simula una corsa dove sei ciclisti si sfidano nella volata finale, visualizzando sullo schermo l'avanzamento di ogni concorrente fino a che tutti giungono al traguardo e la classifica finale.

Descrizione della soluzione

Realizziamo questo sistema definendo per ogni ciclista un thread che si muove con velocità costante per un certo intervallo di tempo (per esempio 10 unità di tempo) e successivamente, in modo casuale, viene modificata tale velocità per un nuovo intervallo di tempo e così via fino a che percorre lo spazio che lo separa dal traguardo del campo di gara.

Iniziamo a codificare la classe `CiclistaInGara` e il suo costruttore:

```
public class CiclistaInGara implements Runnable {
  Ciclista ciclista;
  GaraCiclistica campo;
  int velocita;              // numero di pixel di spostamento al secondo: range 1-4
  Thread t;
  int conta;                 // ogni 10 spostamenti cambio la velocità
  int posizione;             // coordinata X in espressa in pixel

  public CiclistaInGara(Ciclista c, GaraCiclistica g){
    ciclista=c;              // identificativo del ciclista
    campo=g;                 // campo di gara - pista
    conta=0;
    velocita=2;
    t=new Thread(this);
    t.start();
    posizione=0;
  }
```

Il metodo `run()` genera la posizione del ciclista e ne varia la velocità ogni 10 spostamenti: ▶

```
  public void run(){
    //muove il ciclista lungo il percorso, cambiando la velocità
    int dimImmagine=79;      // dimensione dela JPG del corridore
    int dimPista  =960;
    while((ciclista.getCordx()+dimImmagine)<dimPista){  // corsa non finita
      if((conta%10)==0)                                 // ogni 10 spostamenti
        velocita=(int)(Math.random()*4+1);              // modifica la velocità
      ciclista.setCordx(ciclista.getCordx()+velocita);
      conta++;
      try{Thread.sleep(75);}                            // delay
      catch(Exception e){}
      campo.repaint();
    }
    //scrivo in che posizione è arrivato nella classifica finale
    posizione=campo.getPosizione();
    campo.controllaArrivi();
  }
}
```

Codifichiamo una classe `GaraCiclistica` che effettua la gestione della corsa implementando il campo di gara, definendo e inizializzando i singoli thread.

L'intestazione e gli attributi della classe sono i seguenti:

```
import java.awt.*;
import javax.swing.*;

public class GaraCiclistica extends JFrame{
  int posizione;
  Ciclista[] partecipanti;
  CiclistaInGara[] thread_partecipanti;
  Campo pista;
  Graphics offscreen;                    // per la gestione del doppio buffering
  Image buffer_virtuale;
```

Il costruttore definisce la dimensione della pista, definisce e crea i sei partecipanti assegnando a ciascuno di essi la rispettiva corsia.

```
  public GaraCiclistica(){
    //posiziono e disegno il percorso
    super("Gara Ciclistica");
    setSize(1000,645);
    setLocation(10,40);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    pista = new Campo();
    partecipanti= new Ciclista[6];
    thread_partecipanti= new CiclistaInGara[6];
    posizione = 1;
    //aggiungo le immagini dei concorrenti e li associo ai Thread
    int partenza = 15;                   // posizione della prima corsia
    for(int xx=0;xx<6;xx++){
      partecipanti[xx]= new Ciclista(partenza,xx+1);
      thread_partecipanti[xx]= new CiclistaInGara(partecipanti[xx],this);
      partenza=partenza+100;             // posizione verticale  - corsia del ciclista
    }
    // visualizzo la gara
    this.add(pista);
    setVisible(true);
  }
```

Aggiungiamo due metodi che utilizzano le variabili comuni (posizione e arrivati) e quindi li definiamo synchronized:

A il primo assegna la posizione al concorrente che ha appena raggiunto il traguardo;

```
  public synchronized int getPosizione(){
    return posizione++;
  }
```

B il secondo verifica se tutti i concorrenti hanno raggiunto il traguardo e, in tal caso, richiama la visualizzazione della classifica.

```
  public synchronized void controllaArrivi(){
    boolean arrivati=true;
    for(int xx=0;xx<6;xx++){
      if(thread_partecipanti[xx].posizione==0){
        arrivati=false;
      }
    }
    if(arrivati){
      visualizzaClassifica();
    }
  }
```

La classifica viene realizzata dal seguente metodo:

```
public void visualizzaClassifica(){
  JLabel[] arrivi;
  arrivi=new JLabel[6];
  JFrame classifica=new JFrame("Classifica");
  classifica.setSize(500,500);
  classifica.setLocation(280,130);
  classifica.setBackground(Color.BLUE);
  classifica.setDefaultCloseOperation(EXIT_ON_CLOSE);
  classifica.getContentPane().setLayout(new GridLayout(6,1));
  //visualizza l'ordine di arrivo
  for(int xx=1;xx<7;xx++){
    for(int yy=0;yy<6;yy++){
      if(thread_partecipanti[yy].posizione==xx){
        arrivi[yy]=new JLabel(xx+"° classificato ciclista in "+(yy+1)+"° corsia");
        arrivi[yy].setFont(new Font("Times New Roman",Font.ITALIC,20));
        arrivi[yy].setForeground(Color.blue);
        classifica.getContentPane().add(arrivi[yy]);
      }
    }
  }
  classifica.setVisible(true);
 }
```

Concludono la classe `GaraCiclistica` i metodi relativi al disegno sullo schermo mediante la tecnica del doppio buffering necessaria per eliminare lo sfarfallio di immagini in movimento: ▶

```
public void update(Graphics g){
  paint(g);
}

public void paint(Graphics g){
  if (partecipanti != null){
    Graphics2D screen=(Graphics2D)g;
    buffer_virtuale=createImage(1000,645);
    offscreen=buffer_virtuale.getGraphics();
    Dimension d=getSize();
    pista.paint(offscreen);
    for(int xx=0;xx<6;xx++){
      partecipanti[xx].paint(offscreen);
    }
    screen.drawImage(buffer_virtuale,0,30,this);
    offscreen.dispose();
  }
}
```

e il metodo il main, ridotto alla sola creazione di un oggetto della classe: ▶

```
public static void main(String[] a){
  GaraCiclistica a=new GaraCiclistica();
 }
}
```

Riportiamo per completezza la classe `Ciclistica` che si occupa del caricamento e della assegnazione delle immagini ai rispettivi ciclisti:

```
import java.awt.*;
import javax.swing.*;

public class Ciclista extends JPanel{
  int cordx;
  int cordy;
  Image img;

  public Ciclista(int yy,int xx){
    cordx = 0;
    cordy = yy;
    setSize(80,81);
    Toolkit tk=Toolkit.getDefaultToolkit();
    switch (xx){                          // ogni ciclista ha la maglia di colore diverso
       case 1:{img=tk.getImage("bici1.JPG");break;}
       case 2:{img=tk.getImage("bici2.JPG");break;}
       case 3:{img=tk.getImage("bici3.JPG");break;}
       case 4:{img=tk.getImage("bici4.JPG");break;}
       case 5:{img=tk.getImage("bici5.JPG");break;}
       case 6:{img=tk.getImage("bici6.JPG");break;}
    }
    MediaTracker mt=new MediaTracker(this);  // oggetto per la gestione di un numero arbitrario
    mt.addImage(img,1);                      // di immagini in parallelo
    try{mt.waitForID(1);}
    catch(InterruptedException e){}
  }
```

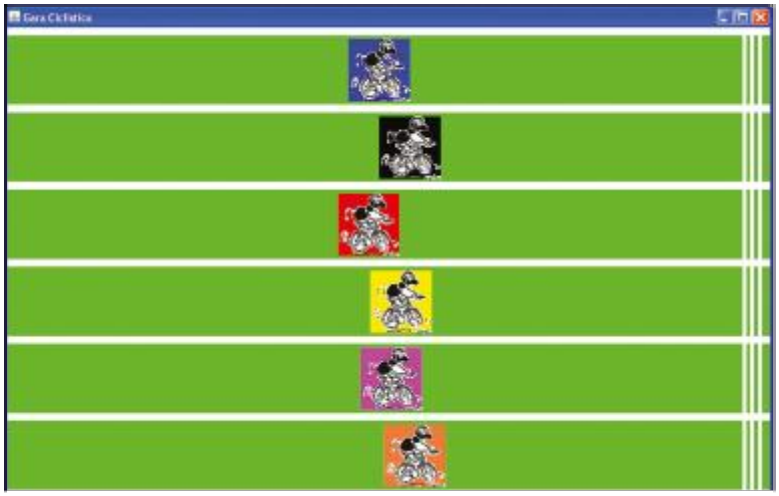
e mette a disposizione i metodi per la gestione della coordinata x del ciclista stesso:

```
  public void setCordx(int n){
    cordx = n;
  }

  public int getCordx(){
    return cordx;
  }

  public void paint(Graphics g){
    g.drawImage(img,cordx,cordy,null);
  }
}
```

Completa il programma la classe `Campo` che disegna il "campo di gara", cioè la pista: ▶

```
import javax.swing.*;
import java.awt.*;

public class Campo extends JPanel {
  public void paint(Graphics g){
    g.setColor(Color.green);
    g.fillRect(0,0,1000,645);
    //Linee laterali
    g.setColor(Color.white);
    g.fillRect(0,0,1000,10);
    g.fillRect(0,100,1000,10);
    g.fillRect(0,200,1000,10);
    g.fillRect(0,300,1000,10);
    g.fillRect(0,400,1000,10);
    g.fillRect(0,500,1000,10);
    g.fillRect(0,600,1000,10);
    //Traguardo
    g.fillRect(960,0,5,645);
    g.fillRect(970,0,5,645);
    g.fillRect(980,0,5,645);
  }
}
```

Nell'immagine seguente è mostrato un momento della corsa.



E una possibile classifica finale:

Prova adesso!

1. Modifica il programma sopra descritto in una corsa con staffetta: ogni ciclista non appena raggiunge il traguardo passa il testimone a un nuovo ciclista che effettua il percorso nel senso opposto fino a raggiungere il punto di partenza che diviene il nuovo traguardo.
2. Si aggiungano i nomi delle squadre e si visualizzi nella classifica il tempo totale e i tempi parziali di percorrenza dei singoli concorrenti.

Esercitazioni di Laboratorio

I THREADS IN JAVA: CONCETTI AVANZATI

Il metodo join()

Quando un processo manda in esecuzione un thread, continua anch'esso la propria evoluzione: si è detto che il main() è un thread e sino a ora lo si è utilizzato per creare e mandare in esecuzione altri thread che successivamente vivono di vita propria (ossia: anche se il processo, o thread, che li ha generati termina, questi continuano la propria esecuzione).
In alcuni casi, potrebbe risultare necessario attendere la terminazione di un thread prima di continuare l'esecuzione, il che implica sospendere un thread in attesa che un secondo thread termini per riprendere quindi l'esecuzione del primo thread. Java mette a disposizione l'operazione join() da eseguirsi sul thread che si intende aspettare.

Vediamo per esempio un main() che manda in esecuzione due thread per poi sospendersi in attesa della terminazione di entrambi: in linea di principio, il codice è il seguente: ▶

```
public static void main(String [] a){
   ...
//avvio l'esecuzione dei due thread
   thr1.start();
   thr2.start();
//attendo la loro terminazione
   thr1.join();
   thr2.join();
```

Ma anche il metodo join() può generare una InterruptedException e quindi deve essere eseguito in un segmento try/catch.
Realizziamo prima il codice eseguito dai due thread: per esempio, facciamo scrivere il loro nome e li facciamo attendere un secondo con il metodo sleep() prima di terminare:

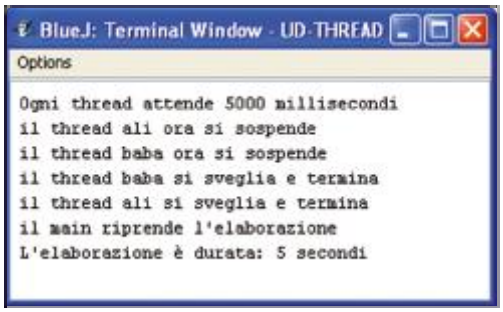
```
public class Aspettali implements Runnable{
 String mionome;
 int tempodormi;
 Aspettali(String chisono,int quanto){
   mionome    = chisono;
   tempodormi = quanto;
 }
 public void run(){
  try{
    System.out.println ("il thread "+mionome+" ora si sospende ");
    Thread.sleep(tempodormi);
    System.out.println ("il thread "+mionome+" si sveglia e termina ");
  }
  catch(InterruptedException e){
    System.out.println(e);
  }
  return;
 }
```

Nel metodo main(), oltre che attivare due thread, inseriamo anche le istruzioni che ci permettono di verificare il tempo totale di esecuzione del main() e quindi calcoliamo il tempo totale di esecuzione del programma. Ci serviamo di un metodo statico della classe System:
static long currenTimeMillis().

Java è stato realizzato e implementato dapprima su sistema Unix, dal quale quindi "eredita" il conteggio del tempo: "l'inizio del mondo Unix" è la mezzanotte del 1° gennaio 1970 e il conteggio del tempo avviene contando i millisecondi trascorsi da tale data.
Il metodo static long currenTimeMillis() fornisce proprio tale valore.

```
public static void main(String [] a){
 long start=0,stop=0,delta=0;
 int quanti = 5000;
 Thread thr1   = new Thread(new Aspettali("ali", quanti));
 Thread thr2   = new Thread(new Aspettali("baba",quanti));
 System.out.println("Ogni thread attende "+quanti+" millisecondi");
 //leggo il tempo alla creazione dei thread
 start = System.currentTimeMillis();
 //avvio l'esecuzione dei due thread
 thr1.start();
 thr2.start();
 try{
   //attendo la loro terminazione
   thr1.join();
   thr2.join();
   //leggo il tempo alla fine dei thread
   stop = System.currentTimeMillis();
   System.out.println("il main riprende l'elaborazione ");
 }
 catch(InterruptedException e){
   System.out.println(e);
 }
 delta = (stop-start)/1000;
 System.out.println("L'elaborazione è durata: "+delta+" secondi");
}
```

Il risultato dell'elaborazione è il seguente:

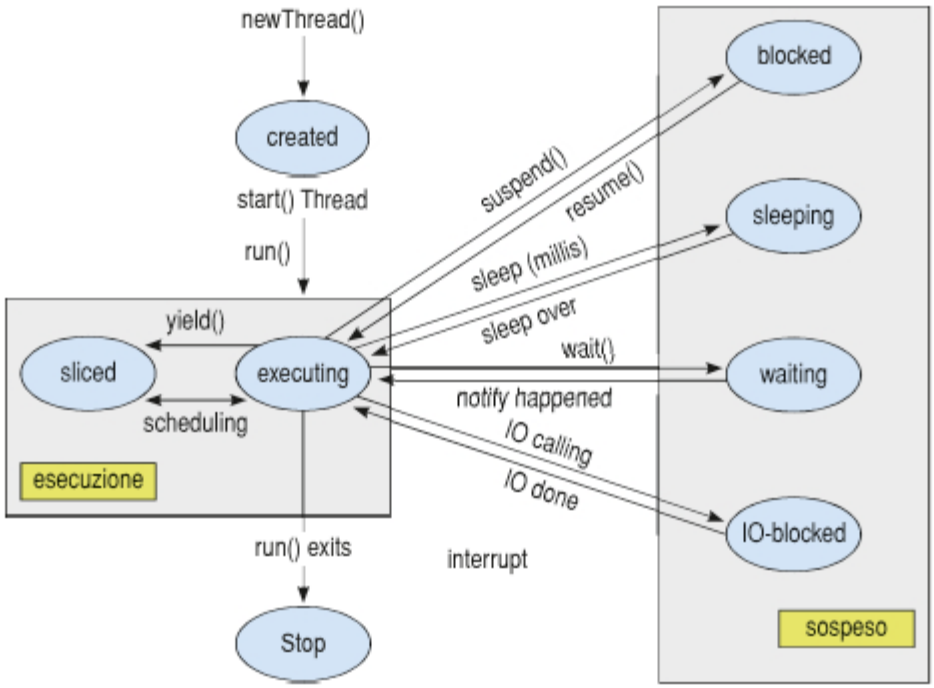


Il tempo totale di attesa è di circa 5 secondi e non 10, che è la somma delle attese dei due processi:
sleep() *non è un'attesa attiva*, cioè non occupa tempo di CPU!

Prova adesso!

Utilizzando questo meccanismo, ovvero ricorrendo a join(), realizza correttamente l'esercizio delle campane facendo cioè in modo che a ogni din segua un don, a ogni don segua un dan e via di seguito!

Nella prossima unità di apprendimento verranno approfonditi gli aspetti legati alla cooperazione tra thread (concorrenza). Nella figura che segue riportiamo il diagramma completo degli stati di un thread indicando i diversi metodi che ne causano il passaggio di stato.



Descriviamoli succintamente:

- start() fa partire l'esecuzione di un thread: la JVM invoca il metodo run() del thread appena creato;
- stop() forza la terminazione dell'esecuzione di un thread;
- suspend() blocca l'esecuzione di un thread in attesa di una successiva operazione di resume(); Non libera le risorse impegnate dal thread;
- resume() riprende l'esecuzione di un thread precedentemente sospeso;
- sleep(long t) blocca per un tempo specificato (time) l'esecuzione di un thread;
- join() blocca il thread chiamante in attesa della terminazione del thread di cui si invoca il metodo;
- yield() sospende l'esecuzione del thread invocante, lasciando il controllo della CPU agli altri thread in coda d'attesa.

Passaggio di parametri a un thread

Il metodo run() che attiva l'esecuzione di un thread non ha parametri formali e quindi non è possibile passare valori attuali per diversificare le esecuzioni, per esempio per fare eseguire operazioni diverse in base a valori diversi. Non è possibile neppure utilizzare il metodo start() dato che anch'esso non ha parametri. La *prima soluzione* si ottiene utilizzando i parametri del metodo costruttore. Vediamo un esempio dove vogliamo che un thread visualizzi i numeri pari e un altro i numeri dispari fino a un valore N.

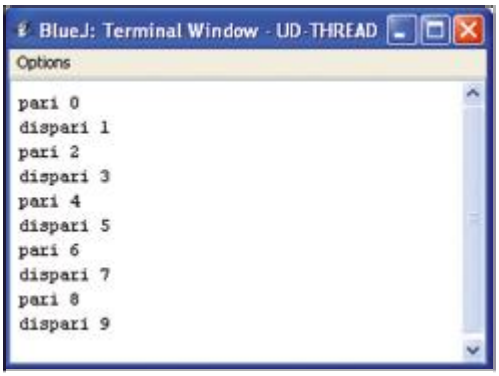
Utilizziamo una variabile booleana (pari) che determinerà se il processo deve stampare i numeri pari o i numeri dispari; il metodo run() è il seguente: ▶

```
public void run(){
  for (int xx=0; xx< massimo; xx++)
    if (pari)          // se è il thread che deve stampare i numeri pari
      if (xx%2==0)     // numero pari
        System.out.println("pari "+xx);
      else             // numero dispari
        System.out.println("dispari "+xx);
}
```

Il primo thread deve essere attivato in un contesto in cui il flag pari viene inizializzato a true mentre il secondo deve avere lo stesso flag con il valore false: inoltre, i due processi devono condividere la variabile N di fine sequenza e quindi necessitano di un ambiente globale comune!
Utilizziamo una variabile d'istanza per ogni dato che intendiamo "personalizzare" nei thread e inizializziamo tali parametri nel metodo costruttore della classe:

```
public static void main(String[] args)  {
  int n = 10 ;
  Thread TP = new PariDispari (n,true);
  Thread TD = new PariDispari (n,false);
  TP.start();
  TD.start();
}
```

L'esecuzione produrrà un output tipo: ▶



La *seconda soluzione* si ottiene incapsulando il thread in una classe nella quale si scrive il metodo costruttore così da rendere possibile il passaggio di un parametro.
Definiamo una classe contatore (riportata di seguito), con il costruttore che riceve due parametri, ne assegna i valori alla variabili locali e crea un thread. ▶

```
class Contatore  extends Thread{
 private int massimo;
 private boolean pari;
 private Thread PD;

 public  Contatore (int finale , boolean pari){
  massimo   = finale;
  this.pari = pari;
  PD        = new Thread(this);
  PD.start();
 }
```

Il metodo run() è lo stesso dell'esercizio precedente: ci rimane ora da definire la classe di prova che è riportata di seguito:

```
public class ProvaConta{
 public static void main(String[] args){
  int n = 10 ;
  Thread TP = new Contatore (n,true);
  Thread TD = new Contatore (n,false);
 }
}
```

L'output è identico al precedente.

Le due modalità portano al medesimo risultato, quindi possono essere impiegate indifferentemente: va tuttavia osservato come in questo secondo caso la dichiarazione del thread è distinta dalla sua attivazione.

Prova adesso!

1. Calcola con N thread differenti i valori del fattoriale di N numeri naturali generati casualmente in [1,10] verificando che non ci siano ripetizioni.
2. Memorizzali in una matrice condivisa e visualizzala sullo schermo.

Priorità di un thread

I thread nella loro evoluzione si comportano come veri e propri processi in termini di occupazione di risorse con il metodo round robin del sistema operativo: ogni singolo thread partecipa alla partizione di tempo con tutti gli altri, acquisendone il relativo *time slice*.
Tutti i thread hanno lo stesso tempo di CPU a disposizione, non essendoci motivazioni particolari per attribuire esecuzioni privilegiate.
In Java ogni thread eredita, all'atto della sua creazione, la priorità del processo padre, ma è possibile modificare le politiche di allocazione delle risorse assegnando ai thread priorità diverse mediante l'inizializzazione di un valore intero compreso tra 1 e 10 (1 è il valore associato al minimo livello di priorità mentre 10 è il massimo).
Il metodo da utilizzare è il seguente:

```
public final void setPriority(int priorita)
```

Il corrispondente metodo get restituisce il valore attualmente impostato nel thread:

```
public int getPriority(int priorita)
```

Sono definite le costanti riportate nella tabella.

Identificatore	Valore
MIN_PRIORITY	1
NORM_PRIORITY	5
MAX_PRIORITY	10

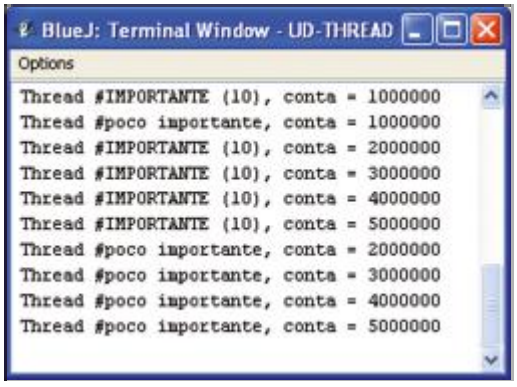
Vediamo un esempio dove due thread con diversa priorità stampano a video il proprio nome:

```
public class Priorita extends Thread{
  private String chisono;

  public Priorita (String nome){ setChisei(nome);}

  public String getChisei()     { return chisono;}

  public void setChisei(String nome) { chisono=nome;}

  public void run(){
    int conta =0;
    while (conta < 5000000) {
     conta++;
     if ((conta % 1000000) == 0)
       System.out.println("Thread #" + chisono + ", conta = " + conta);
     }
  }
```

Il metodo main() passa come parametro ai due processi la loro importanza settandone uno al valore massimo (MAX_PRIORITY) e uno al minimo (MIN_PRIORITY):

```
public static void main (String [] args)
{
  Thread TA = new Priorita ("IMPORTANTE");
  Thread TB = new Priorita ("poco importante");
  TA.setPriority(Thread.MAX_PRIORITY);
  TB.setPriority(Thread.MIN_PRIORITY);
  TA.start();
  TB.start();
}
```

Un'esecuzione genera il seguente output:



Un caso particolare potrebbe essere quello in cui tanti thread ad alta priorità ostacolano l'evoluzione di un thread a bassa priorità: affronteremo tale argomento quando ci occuperemo della cooperazione tra i thread. È comunque disponibile un metodo (yield()) con cui si può temporaneamente ridurre la priorità di un thread per fare in modo che anche quelli con bassa priorità possano evolvere.

Un particolare thread a bassa priorità è il *garbage collector*: esso rientra in una famiglia di thread particolari, detti *thread deamon*, che sono servizi di sistema attivati quando la CPU ha un basso numero di risorse utilizzate.

Prova adesso!

1. Realizza un programma in Java che attivi tre contatori indipendenti e ne visualizzi i valori (totali o parziali) in un intervallo massimo di 5 secondi.
2. Il contatore deve incrementare una proprietà interna (pubblica o privata) ed effettuare una visualizzazione a ogni ciclo.
3. Quindi a ogni contatore assegna una priorità iniziale diversa.
4. Analizzino i risultati proponendo le relative considerazioni.

Thread e animazioni

È possibile utilizzare i thread per animare disegni e immagini.
Realizziamo una applet dove un semplice orologio digitale scandisce i secondi.

```
import java.awt.Graphics;
import java.awt.Font;
import java.util.Date;
import java.applet.*;
public class Orodig extends java.applet.Applet implements Runnable{
  Font miofont = new Font("TimesRoman",Font.BOLD,24);
  Date miadata;
  Thread miothread;
  public void run(){
    while (true){
      miadata = new Date();
      repaint();
      try   { Thread.sleep(1000); }
      catch (InterruptedException e){}
    }
  }
  public void paint( Graphics g){
     g.setFont(miofont);
     g.drawString(miadata.toString(),10,50);
  }
  public void start(){
    if( miothread == null ){
      miothread = new Thread(this);
      miothread.start();
    }
  }
}
```

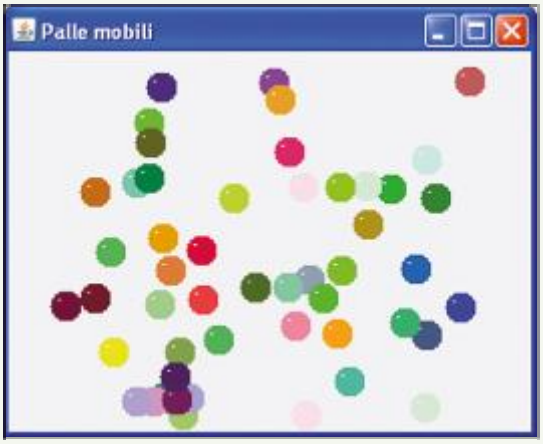
Il funzionamento è evidente: il thread si sospende per un secondo e, quando riprende, legge il nuovo valore dell'orologio di sistema ed esegue repaint()visualizzando la nuova ora sullo schermo. Viene cioè usato il metodo sleep() come sincronizzatore per la visualizzazione dei fotogrammi. Il risultato è quello della figura che segue.

Naturalmente, con disegni più complessi avremo il problema dello sfarfallio; un processo completo è riportato negli esempi, nei quali viene applicata la tecnica del doppio buffering (◀ **double-buffering** ▶.

◀ **Double-buffering** The traditional notion of double-buffering in Java applications is fairly straightforward: create an offscreen image, draw to that image using the image's graphics object, then, in one step, call drawImage using the target window's graphics object and the offscreen image. You may have already noticed that Swing uses this technique in many of its components, usually enabled by default, using the setDoubleBuffered method.
(Java tutorial http://docs.oracle.com/javase/tutorial/extra/fullscreen/doublebuf.html)

Prova adesso!

Realizza un programma che generi casualmente un numero di thread dove ciascuno disegna una palla colorata e la anima, facendola rimbalzare sui bordi, come nell'esempio di figura.



Puoi confrontare la tua soluzione con quella riportata nella classe PalleMobili.

Zoom su...

DOPPIO BUFFERING

La soluzione ottimale per l'eliminazione dello sfarfallio nell'animazione di disegni e immagini è la tecnica che prende il nome di bufferizzazione doppia o doppio buffering.

> Questa tecnica consiste nel creare fuori dallo schermo (in un buffer) il fotogramma che segue quello che è visualizzato per poi sostituirlo rapidamente mentre nel buffer viene creato quello successivo.

È un accorgimento di grande effetto che elimina completamente lo sfarfallio: per la sua realizzazione, però, rende più complicate le operazioni necessarie alla preparazione del disegno richiedendo alcuni passaggi supplementari:

1. dichiarazione di una variabile immagine (oggetto della classe Image) e di un contesto grafico (oggetto della classe Graphics). L'insieme delle due variabili è il "doppio buffer";
2. creazione di una immagine grande quanto la finestra e inizializzazione del contesto grafico;
3. ridefinire il metodo update() in modo che esegua solo la chiamata a paint() e non cancelli lo schermo
4. disegnare usando il nuovo contesto grafico (e non quello fornito come parametro dal metodo paint());
5. ricopiare il buffer nella finestra (ovvero disegnare l'immagine creata usando il nuovo contesto grafico).

2 COMUNICAZIONE E SINCRONIZZAZIONE

UNITÀ DI APPRENDIMENTO

OBIETTIVI

- Conoscere il modello ad ambiente globale
- Conoscere il modello ad ambiente locale
- Individuare le tipologie di errori nei processi paralleli
- Comprendere l'esigenza di sincronizzazione
- Definire ed utilizzare i semafori di basso livello e spin lock()
- Comprendere il concetto di indivisibilità di una primitiva
- Sapere il funzionamento dei semafori di Dijkstra
- Avere il concetto di regione critica e di mutua esclusione
- Sapere la differenza tra interleaving e overlapping
- Comprendere le condizioni di Bernstein
- Avere il concetto starvation e di deadlock
- Sapere in cosa consistono le proprietà di safety, di fairness e di liveness

ATTIVITÀ

- Risolvere le situazioni di starvation
- Risolvere le situazioni di deadlock
- Utilizzare gli strumenti di sincronizzazione per thread in C sotto Linux
- Risolvere i problemi produttore/consumatore in C
- Utilizzare le condition variable in C
- Implementare i monitor in C
- Utilizzare gli strumenti di sincronizzazione per thread in C sotto Linux
- Risolvere il problema dei filosofi in C
- Risolvere i problemi produttore/consumatore in Java
- Implementare i monitor in Java
- Risolvere i problemi produttore/consumatore con i monitor in Java
- Risolvere il problema dei filosofi con il linguaggio Java

LEZIONE 1

LA COMUNICAZIONE TRA PROCESSI

IN QUESTA LEZIONE IMPAREREMO...

- il modello ad ambiente globale o a memoria condivisa
- il modello ad ambiente locale o a scambio di messaggi

Comunicazione: modelli software e hardware

In un ambiente di processi concorrenti le situazioni e le possibilità nelle quali i processi devono comunicare tra loro sono molteplici.
Innanzi tutto la possibilità può essere offerta dalla tipologia della architettura hardware e, come già detto, noi ci occuperemo solo di architetture monoprocessore, di quelle cioè che nella classificazione di Flynn si basano sul modello Von Neumann (modello sequenziale con una sola capacità di esecuzione SISD Singol Instruction Singol Data).

Zoom su...

CLASSIFICAZIONE DI FLYNN

Ricordiamo la classificazione di Flynn che definisce i modelli di esecuzione considerando la molteplicità dei flussi di dati e di esecuzione ottenendo quattro combinazioni e quindi quattro modelli differenti:

Data stream/Instruction Stream	Unico flusso dati	Flussi di dati multipli
Unico flusso di istruzioni	SISD (von Neuman)	SIMD (sistemi vettoriali)
Flussi Multipli di istruzioni	MISD (pipeline)	MIMD (macchine multiple)

Per esempio, abbiamo:
- Single Instruction Multiple Data (SIMD);
- Multiple Instruction Multiple Data (MIMD).

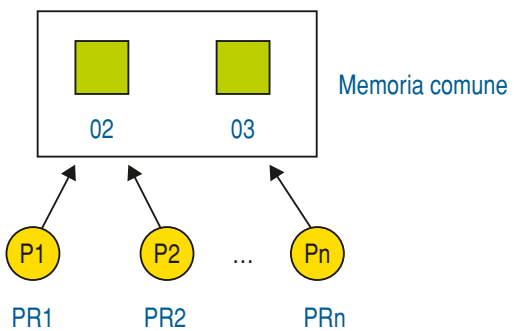
Ma anche in questo caso, cioè considerando solo macchine monoprocessore, in un sistema di rete è connessa una molteplicità di macchine di von Neumann che lavorano su flussi di esecuzione paralleli e quindi interagiscono tra loro condividendo risorse comuni.
Inoltre su una macchina monoprocessore il SO multitasking permette l'esecuzione contemporanea di più processi che competono per accedere alle risorse comuni.

Possiamo individuare due modelli di interazione concorrente a prescindere dalla soluzione hardware, cioè sia che stiamo analizzando una architettura distribuita oppure una situazione di multitask su macchina SISM:
- modello a memoria comune (ambiente globale, global environment);
- modello a scambio di messaggi (ambiente locale, message passing).

Entrambi i modelli si basano sul concetto di interazione tra i due elementi che costituiscono il sistema: i processi interagiscono per entrare in possesso (utilizzare) delle risorse (oggetti).

Modello a memoria comune (ambiente globale, global environment)

Il modello a memoria comune trova naturale impiego nelle architetture in cui esiste un'unica memoria comune a tutti i processi (o processori), per esempio su macchine monoprocessore con processi multitasking.



È invece complesso (e costoso) condividere memoria nei sistemi di elaborazione distribuiti, nei quali si preferisce utilizzare il meccanismo a scambio di messaggi.

Allocazione delle risorse ai processi

Il modello a memoria comune è il solo caso nel quale possono verificarsi problemi per l'accesso a essa da parte di due (o più) processi che ne potrebbero richiedere l'attribuzione contemporaneamente.

Il sistema operativo associa a ogni risorsa un apposito *gestore di risorsa* (o *allocatore*), cioè un segmento di codice che gestisce tutte le richieste fatte dai diversi processi che necessitano di utilizzare la specifica risorsa della quale ne è l'*allocatore*, cioè può assegnarla a un processo o negarla a seconda dello stato in cui si trova (per esempio può essere "occupata", cioè in uso a un altro processo).

Sostanzialmente possiamo riassumere i suoi compiti in:
1. deve mantenere aggiornato lo *stato di allocazione della risorsa*;
2. deve *fornire i meccanismi* ai processi che hanno il diritto di utilizzare tale risorsa di accedervi, prenderne possesso, operare su di essa e alla fine del suo utilizzo di "liberarla" per gli altri processi;
3. deve *implementare la strategia di allocazione* della risorsa definendo a quale processo e per quanto tempo assegnare la risorsa.

In generale un allocatore si interfaccia con i processi mediante due procedure:
- al momento in cui un processo ha bisogno di una risorsa fa una richiesta di assegnazione (*acquisizione*) mediante la prima procedura;
- la seconda viene chiamata dal processo che sta utilizzando la risorsa quando termina di averne bisogno e intende "renderla libera" restituendola al sistema operativo (*rilascio*).

ESEMPIO 1

Pensiamo per esempio a un processo che necessita di stampare un documento: in questo caso il gestore della risorsa è lo spooler di stampa che all'atto di una richiesta la pone in una coda di attesa e la soddisfa quando un processo termina di stampare e lascia libera la stampante.

Tipologie di allocazione delle risorse nel modello ad ambiente globale

Come risorsa intendiamo qualunque oggetto, fisico o logico, di cui un processo necessita per portare a termine il suo compito: le risorse vengono raggruppate in classi e una classe identifica le risorse che hanno in comune tutte e sole le operazioni che un processo può eseguire per operare su ciascun componente di quella classe.

Anche una istanza di una *struttura dati* allocata nella *memoria comune* è una risorsa e, come vedremo, sarà proprio tramite questa che più processi o thread si scambiano informazioni.

Possiamo classificare le risorse in private e comuni, in base al modello di macchina e al loro tipo di allocazione.

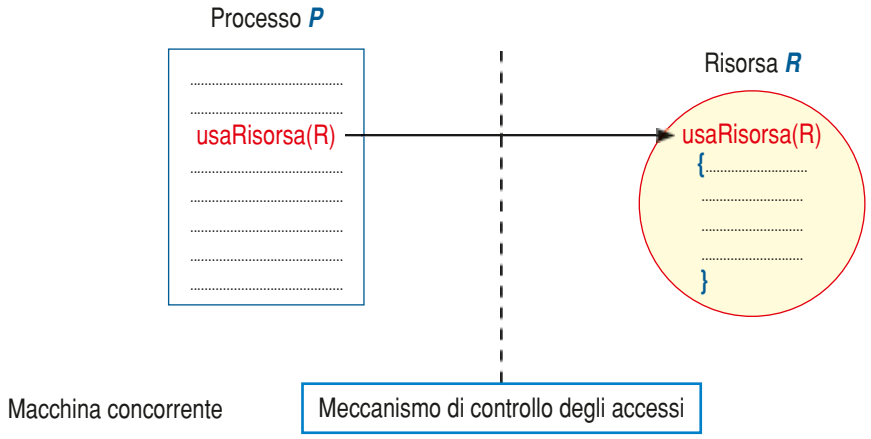
	Risorse dedicate (visibili in ogni istante da un solo processo)	**Risorse condivise** (visibili in ogni istante anche da più processi contemporaneamente)
risorse allocate staticamente	risorse private	risorse comuni
risorse allocate dinamicamente	risorse comuni	risorse comuni

Per ogni risorsa il relativo gestore definisce istante per istante i processi che hanno il diritto di operare su di essa.

Le risorse allocate staticamente vengono definite prima che il programma inizi la propria esecuzione e quindi il loro gestore è il programmatore.

Nel caso di risorse dedicate, sia allocate staticamente che dinamicamente, non è necessario nessun controllo da parte del programmatore per quanto riguarda la sincronizzazione dato che sono di utilizzo esclusivo di un singolo processo e vengono assegnate e gestite dal sistema operativo.
Nel caso di risorse condivise il programmatore, utilizzando i costrutti del linguaggio di programmazione, stabilisce le regole di visibilità e quindi quali processi possono operare sui dati comuni, in quali istanti possono accedere alla risorsa, definendo le modalità di sincronizzazione (come vedremo nelle prossime lezioni con la gestione delle sezioni critiche).

Gli accessi a una risorsa condivisa devono avvenire in modo non divisibile: le funzioni che utilizzano un dato condiviso (sezioni critiche) devono essere programmate utilizzando i meccanismi di sincronizzazione offerti dal linguaggio di programmazione e supportati dalla macchina concorrente.



Vediamo nelle diverse situazioni come interagiscono i processi, cioè quando *competono* e quando *cooperano* nelle due situazioni di allocazione *statica* e *dinamica*.

MUTUA ESCLUSIONE

L'accesso a una risorsa si dice mutualmente esclusivo se a ogni istante, al massimo un processo può accedere a quella risorsa.

Competizione

Nel caso di risorse condivise e allocate staticamente la competizione tra processi avviene al momento dell'accesso alla risorsa: è necessario garantire l'accesso esclusivo (mutua esclusione) alla stessa e il compito è del programmatore che, utilizzando le primitive disponibili dal linguaggio di programmazione, scrive le funzioni di accesso in modo che solo un processo alla volta possa utilizzare il dato condiviso.

Nel caso di risorse dedicate e allocate dinamicamente la responsabilità di gestione è demandata al gestore e la competizione tra processi avviene al momento della richiesta di utilizzo indirizzata al gestore stesso che provvederà a gestirle in mutua esclusione.

Cooperazione

Nel caso di risorse condivise e allocate staticamente la cooperazione tra processi avviene utilizzando tale risorsa per scambiarsi le informazioni, cioè un processo (o più processi) scrivono un dato nella risorsa (produttori) e un altro processo (o più processi) leggono successivamente dalla risorsa condivisa (consumatori).

Nel caso di risorse dedicate e allocate dinamicamente l'unica possibilità di cooperazione si ha se una risorsa assegnata a un processo viene assegnata a un secondo processo quando il primo termina di utilizzarla: il gestore deve essere al corrente della cooperazione in modo da non azzerare il contenuto "prodotto" dal primo processo prima che venga "consumato" dal secondo processo.

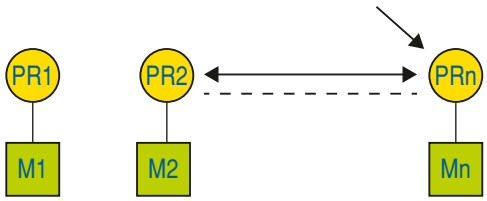
Il caso più complesso è quello delle risorse condivise e allocate dinamicamente dove è necessario combinare le situazioni sopra presentate: parte dei compiti è di competenza del programmatore e parte demandata al gestore della risorsa, cioè ai componenti del sistema operativo.

Modello a scambio di messaggi (ambiente locale, message passing)

Nel modello ad ambiente locale ogni processo può accedere esclusivamente alle risorse allocate nella propria memoria (virtuale) locale che non può essere modificata direttamente dagli altri processi.

Non avendo una memoria condivisa, i processi non possono utilizzare la memoria per il coordinamento delle loro attività e lo strumento di comunicazione e sincronizzazione diventa lo *scambio di messaggi*.

Il modello a scambio di messaggi rappresenta la naturale **astrazione di un sistema privo di memoria comune**, in cui a ciascun processore è associata una memoria privata.



Il sistema è visto come un insieme di processi, ciascuno operante in un ambiente locale che non è accessibile a nessun altro processo.

Esistono due possibili modalità per implementare questo modello:

- **A** utilizzare linguaggi che prevedono costrutti espliciti per realizzare lo scambio di messaggi, come il CSP (Communicating Sequential Processes) proposto da ◀ **Tony Hoare** ▶;
- **B** utilizzare la "chiamata di procedura remota", come il DP (Distributed Processes), proposto da ◀ **Brinch Hansen** ▶.

◀ **Hoare & Hansen** Hoare is a British computer scientist best known for the development (in 1960, at age 26) of Quicksort, He also developed the formal language Communicating Sequential Processes (CSP) to specify the interactions of concurrent processes (including the dining philosophers problem).
Hansen is a pioneer in the development of operating system principles and parallel programming languages(ex. the parallel programming languages Concurrent Pascal) Dijkstra, Hoare, and Brinch Hansen suggested a parallel programming concept in 1971: the monitor, ▶

È inoltre necessario effettuare una classificazione dei modelli a scambio di messaggi: per esempio possiamo distinguere tra comunicazione sincrona e asincrona:

- **A** nel caso di comunicazione asincrona la comunicazione da parte del processo mittente avviene senza che questo rimanga in attesa di una risposta da parte del processo destinatario;
- **B** nel caso di comunicazione sincrona lo scambio di informazioni può avvenire solo se mittente e destinatario sono pronti a "parlarsi" e quindi è necessario che si sincronizzino, e questa interazione prende il nome di ◀ **"rendez-vous"** ▶:
  - stretto: se si limita alla trasmissione di un messaggio dal mittente al destinatario;
  - esteso: se il destinatario, una volta ricevuto il messaggio, deve inviare una risposta al mittente.

◀ **"Rendez-vous"** A rendez-vous is a synchronization or meeting point between the task calling another task's entry and the called task. The first task to the rendez-vous will suspend until another task gets to the same rendezvous. ▶

È possibile effettuare una seconda classificazione dei modelli a scambio di messaggi, distinguendo tra comunicazione asimmetrica e simmetrica:

- **A** nel caso di comunicazione asimmetrica il mittente nomina esplicitamente il destinatario ma questo non nomina esplicitamente il mittente;
- **B** nel caso di comunicazione simmetrica entrambi si nominano in modo esplicito.

Possiamo quindi avere per esempio queste due classiche situazioni:

- comunicazione di tipo simmetrico e sincrono a rendez-vous stretto tipico del CSP;
- comunicazione di tipo asimmetrico e sincrono a rendez-vous esteso tipico del DP.

Le applicazioni di queste proposte trovano il loro utilizzo fondamentalmente nei sistemi multiprocessori: un esempio tipico è quello noto come modello client-server.

Modello client-server

Ogni risorsa del sistema è accessibile a un solo processo che prende il nome di processo servitore (o server) e quando un processo deve utilizzarla (processo cliente) non può accedervi direttamente ma deve chiedere al processo server di effettuare lui stesso le operazioni desiderate sulla risorsa e di comunicargli successivamente l'esito delle elaborazioni.

Il meccanismo che viene quindi utilizzato dai processi è quello prima descritto di **scambio di messaggi** e solo tramite questi sono possibili tutti i tipi di interazione tra i processi.

Definiamo quindi:

SERVITORE/CLIENTE

Servitore: entità computazionale in grado di eseguire una specifica prestazione per altre entità (in grado cioè di offrire un servizio).
Cliente: entità computazionale che richiede a un servitore l'esecuzione di una specifica prestazione.

Il modello cliente-servitore è il principio su cui è possibile costruire applicazioni distribuite: tipico esempio è la rete Internet, dove un cliente risiede in un nodo e la risorsa è disponibile in un altro; il cliente deve fare la richiesta a una servitore locale della risorsa per poterla utilizzare, come per esempio per accedere a un database o a un file, e il servente esegue sulla risorsa le richieste comunicando l'esito attraverso messaggi (che generalmente sono pagine HTML o trasferimento di file).

Verifichiamo le conoscenze

Esercizi a scelta multipla

1 Il parallelismo dell'esecuzione di processi concorrenti:

a) è solo virtuale
b) è solo reale
c) è reale su sistemi multiprocessing e virtuale su sistemi multiprocessor
d) è reale su sistemi multiprocessor e virtuale su sistemi multiprocessing
e) nessuna delle precedenti

2 Nella classificazione di Flynn il personal computer è

a) una macchina SISD
b) una macchina SIMD
c) una macchina MISD
d) una macchina MIMD

3 Quali dei seguenti accoppiamenti è esatto?

a) modello a memoria comune o ambiente locale
b) modello a memoria comune o ambiente globale
c) modello a scambio di messaggi o ambiente locale
d) modello a scambio di messaggi o ambiente globale

4 I compiti del gestore della risorsa sono (indicare quello errato):

a) deve mantenere aggiornato lo stato di allocazione della risorsa
b) deve sospendere i processi che utilizzano la risorsa per molto tempo
c) deve implementare la strategia di allocazione della risorsa
d) deve fornire i meccanismi ai processi che hanno il diritto di utilizzare tale risorsa

5 Le risorse allocate staticamente sono:

a) sempre private
b) private se dedicate
c) sempre comuni
d) comuni se condivise

6 Le risorse allocate dinamicamente sono:

a) sempre private
b) private se dedicate
c) sempre comuni
d) comuni se condivise

6 Nel modello a scambio di messaggi le risorse comuni:

a) sono allocate staticamente
b) sono allocate dinamicamente
c) non ci sono risorse condivise
d) sono allocate in maniera dipendente dall'architettura

7 Quali tra i seguenti sono tipi di comunicazione nei modelli a scambio di messaggi?

a) sincrono a rendez-vous stretto
b) sincrono a rendez-vous stretto
c) asincrono a rendez-vous esteso
d) asincrono a rendez-vous esteso

Test vero/falso

1 L'acronimo SIMD indica Single Instruction Multiprocessor Data. V F
2 È comodo condividere memoria nei sistemi di elaborazione distribuiti. V F
3 Il sistema operativo associa a ogni risorsa un apposito gestore di risorsa. V F

4 Si può considerare risorsa anche una istanza di una struttura dati allocata nella memoria comune. V F
5 Il gestore definisce istante per istante i processi che hanno il diritto di operare su di essa. V F
6 Il gestore delle risorse allocate staticamente è il programmatore. V F
7 Gli accessi a una risorsa condivisa possono avvenire in modo non divisibile. V F
8 L'accesso si dice mutualmente esclusivo se solo un processo può accedere a quella risorsa. V F
9 Il caso più complesso di gestione è quello di risorse condivise e allocate dinamicamente. V F
10 Il linguaggio come il CSP proposto da Hansen prevede costrutti espliciti per realizzare lo scambio di messaggi. V F
11 Nella comunicazione simmetrica il mittente nomina esplicitamente il destinatario e viceversa. V F

Esercizi a completamento

1 Possiamo individuare due modelli di ..... a prescindere dalla soluzione hardware, cioè sia che stiamo analizzando una ..... oppure una situazione di ..... su macchina .....;
- modello a .....;
- modello a .....

2 Entrambi i modelli si basano sul concetto di ..... tra i due elementi che costituiscono il sistema: i ..... interagiscono per entrare in possesso (utilizzare) delle ..... (oggetti).

3 Gli accessi a una ..... devono avvenire in modo .....: le funzioni che utilizzano un dato condiviso (.....) devono essere programmate utilizzando i meccanismi di ..... offerti dal linguaggio di programmazione e supportati dalla macchina concorrente.

4 L'accesso a una risorsa si dice ..... se a ogni istante, al massimo un processo può accedere a quella risorsa.

5 Nel caso di risorse ..... e ..... la competizione tra processi avviene al momento dell'accesso alla risorsa: è necessario garantire l'accesso esclusivo (.....).

6 Nel caso di risorse ..... e ..... la cooperazione tra processi avviene utilizzando tale risorsa per scambiarsi le informazioni

7 Nel caso di comunicazione ..... la comunicazione da parte del processo mittente avviene ..... che questo rimanga in attesa di una risposta da parte del processo destinatario;

8 Nel caso di comunicazione ..... lo scambio di informazioni può avvenire solo se mittente e destinatario sono pronti a "parlarsi" e quindi è necessario che si ....., e questa interazione prende il nome di "".
- .....: se si limita alla trasmissione di un messaggio dal mittente al destinatario;
- .....: se il destinatario, una volta ricevuto il messaggio, deve inviare una risposta al mittente.

LEZIONE 2

LA SINCRONIZZAZIONE TRA PROCESSI

IN QUESTA LEZIONE IMPAREREMO...

- le tipologie di errori nei processi paralleli
- le motivazioni della sincronizzazione
- le proprietà richieste ai programmi concorrenti

Errori nei programmi concorrenti

La programmazione concorrente nasconde maggiori insidie della normale programmazione monoprogrammata in quanto introduce la possibilità di commettere errori dipendenti dal tempo, nei confronti dei quali le normali tecniche di debugging non sono efficaci dato che, oltre alla correttezza logica, ai programmi è anche richiesta la correttezza temporale.

È molto diverso effettuare il testing di un programma concorrente rispetto a uno sequenziale:

- il programma sequenziale produce diversi risultati ma solo in funzione di dati di input diversi e il programmatore, mediante casi di prova, può verificare i comportamenti del programma confrontando gli output con i risultati attesi: non è possibile in questo modo dimostrare la correttezza totale del programma, ma se viene accuratamente scelto l'insieme dei dati di prova possiamo avere buone garanzie sulla correttezza del nostro lavoro;
- nei programmi concorrenti, oltre agli errori sequenziali eliminabili come prima descritto, sono possibili gli errori legati ai tempi di esecuzione e di schedulazione nella CPU che non si possono determinare ed eliminare tramite testing ma devono essere evitati tramite una programmazione particolarmente accurata, individuando "dove il codice" potrebbe essere causa di possibili situazioni di errore, che quasi sempre è connesso con la condivisione e comunicazione di dati tra processi concorrenti.

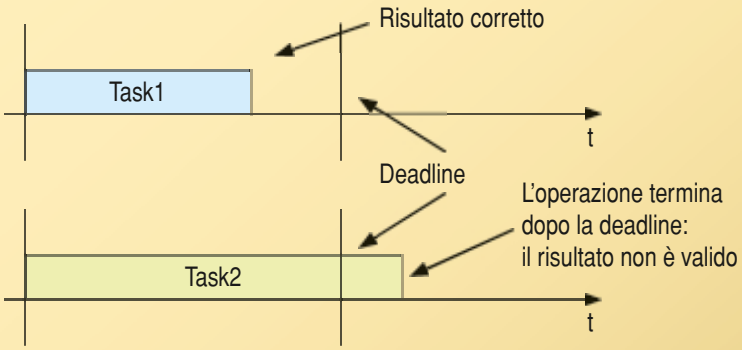
La soluzione estrema che permette di eliminare i problemi dipendenti dal tempo è quella di introdurre ritardi nelle elaborazioni tali che possano escludere la generazione di problemi dovuti alla loro interazione; tuttavia questa non è una soluzione, è un "artificio non accettabile" oltre che rischioso in quanto non offre nessuna certezza che renda l'esecuzione corretta.

La maggior parte dei sistemi concorrenti ha inoltre la necessità di sfruttare al massimo le potenzialità del sistema di calcolo e richiede di conciliare l'affidabilità del software all'esigenza di massimizzare l'efficienza di elaborazione, soprattutto nei sistemi a ◀ hard real time ▶, dove è assolutamente indispensabile il rispetto delle ◀ deadlines temporali ▶ (timing constraint).

Abbiamo due vincoli da soddisfare indicati generalmente come "correttezza temporale":

Ⓐ determinismo: i risultati devono essere uguali per ogni esecuzione, indipendentemente dalla sequenza di schedulazione;

Ⓑ timing constraint: i risultati devono essere prodotti entro certi limiti temporali fissati (deadlines) (specifica dei sistemi real time).



◀ Deadlines temporali È l'istante temporale entro cui il processo deve terminare la propria esecuzione e produrre un risultato. ▶

◀ **Hard real time** A hard real-time system (also known as an immediate real-time system) is hardware or software that must operate within the confines of a stringent deadline. Examples of hard real-time systems include components of pacemakers, anti-lock brakes and aircraft control systems. ▶

Gli errori dipendenti dal tempo sono causati da una scorretta sincronizzazione dei processi e costituiscono una particolare categoria di errori: si verificano in corrispondenza a determinate velocità relative dei processi e non si riproducono quindi necessariamente riavviando il sistema con le stesse condizioni iniziali.

Riassumiamo le caratteristiche degli errori dipendenti dal tempo:

- irriproducibili: possono verificarsi con alcune sequenze e non con altre;
- indeterminati: esito ed effetti dipendono dalla sequenza;
- latenti: possono presentarsi solo con sequenze rare;
- difficili da verificare e testare: perché le tecniche di verifica e testing si basano sulla riproducibilità del comportamento.

È quindi di fondamentale importanza la scelta di tecniche corrette di sincronizzazione.

Se una risorsa è allocata come dedicata non è necessaria la sincronizzazione mentre se una risorsa è condivisa è necessario assicurare che gli accessi avvengano in modo non divisibile, cioè che le operazioni che un processo deve effettuare sulla risorsa, per esempio l'aggiornamento di un dato, non vengano interrotte neppure dallo scheduler ma si possa garantire l'accesso in mutua esclusione finché il processo stesso non decide di rilasciarla al termine del suo utilizzo in modo da rendere disponibile il risultato dell'elaborazione quando questo è significativo.
L'insieme delle operazioni che devono essere ininterrompibili devono essere programmate come sezioni critiche utilizzando i meccanismi di sincronizzazione offerti dal linguaggio di programmazione.

Zoom su...

APPLICAZIONI REAL TIME

Alcune classiche applicazioni real time sono le macchine automatiche, le automobili, i computer, i sistemi di telecomunicazione, le centrali elettriche, l'avionica ecc. che sono classificate tra hard real time e soft real time: la differenza è che nelle applicazioni hard real time la deadline temporale è assolutamente inderogabile mentre in quelle soft anche se viene superata non causa effetti dannosi ed è quindi tollerabile (per esempio, se si spegne la caldaia qualche minuto dopo la deadline non succede nulla di irreparabile mentre se il freno dell'automobile interviene qualche secondo dopo che viene schiacciato, il ritardo, anche se piccolo, potrebbe essere fatale!).

Spesso si confonde il concetto di hard e soft real time con quello di real time "stretto" e "largo":
- real time "stretto": il tempo a disposizione per le elaborazioni richieste è scarso e quindi il sistema deve essere il più efficiente possibile;
- real time "largo": non ci sono vincoli temporali particolari quindi si possono eseguire le operazioni senza "stressare" il sistema.

Definizioni e proprietà

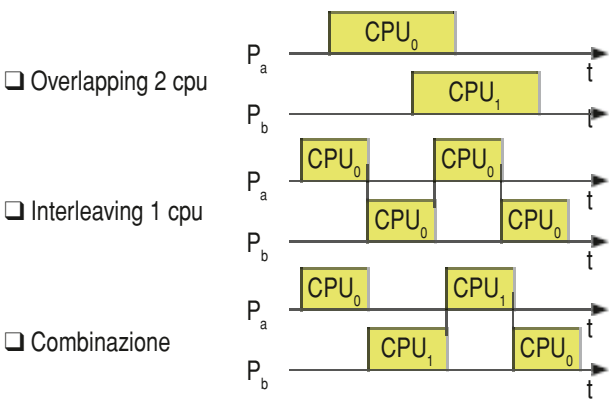
Prima di procedere con lo studio dei meccanismi che permettono di realizzare la mutua esclusione è necessario introdurre alcune definizioni e rivedere quelle incontrate sino a questo punto della trattazione per completarne la formalizzazione.

Interleaving e overlapping

Innanzi tutto per avere la concorrenza è necessario che due programmi siano eseguiti in parallelo, che può essere *parallelismo reale*, nel caso di più processori, *o apparente*, in macchine multiprogrammate monoprocessore.

Nella macchine con un singolo processore i processi sono "*alternati nel tempo*" ma con velocità tali da dare l'impressione di avere un multiprocessore: in questo caso si definisce interleaving la situazione di alternanza casuale che possono avere i processi a causa delle diverse modalità di schedulazione (che può portare a errori dipendenti dal tempo).
Nei sistemi multiprocessore più processi vengono eseguiti simultaneamente su processori diversi e quindi sono "*sovrapposti nel tempo*": in questo caso si definisce overlapping la sovrapposizione temporale di processi.

Potrebbe anche esserci una situazione in cui sono presenti più macchine, ciascuna multitasking e quindi avere una combinazione delle due situazioni sopra descritte: ▶



Condizioni di Bernstein

Scrivere quindi un programma concorrente non è facile: abbiamo visto come è possibile rappresentare un programma sequenziale in termini di precedenze ma non sappiamo che caratteristiche devono avere due operazioni che possono essere eseguite in parallelo senza che generino errori dipendenti dal tempo.

I vincoli che devono soddisfare due istruzioni per essere eseguite concorrentemente sono le condizioni di Bernstein.

Prima di elencarle è necessario introdurre il concetto di dominio e di rango di una procedura, che si ottiene dall'unione dei domini e ranghi delle singole istruzioni.

DOMINIO E RANGO DI UNA ISTRUZIONE O PROCEDURA

Indicando A, B, …, X, Y, … una variabile o, più generalmente, un'area di memoria, una istruzione K:

- si ottiene da una o più aree di memoria, che indichiamo come domain(K) (dominio di K);
- modifica il contenuto di una o più aree di memoria, che indichiamo con range(K) (rango di K).

ESEMPIO 2 ***Dominio e rango***

La seguente procedura P:

```
procedura P
inizio
   X ← A - X;
   Y ← A * B;
fine
```

utilizza tre variabili, A, B e Y, quindi si ha domain(P) = {A, B, X}
modifica due variabili, X e Y, quindi si ha range(P) = {X, Y}

Possiamo ora definire le:

CONDIZIONI DI BERNSTEIN

Due istruzioni i_a e i_b possono essere eseguite concorrentemente se valgono le seguenti condizioni, dette condizioni di Bernstein:

- range(istruzione A) ∩ range(istruzione B) = ∅
- range(istruzione A) ∩ domain(istruzione B) = ∅
- domain(istruzione A) ∩ range(istruzione B) = ∅

Non si impone alcuna condizione sulla intersezione dei domini delle due istruzioni.

ESEMPIO 3 *Verifica delle condizioni di Bernstein*

Vediamo tre semplici esempi dove analizziamo due istruzioni alla volta:

1

Istruzione	Domain(istruzione)	Range(istruzione)
X ← Y + 5;	Y	X
X ← Y – 3;	Y	X

violano la condizione 1:
- range(A) ∩ range(B) = X che è diverso dall'insieme vuoto ∅

2

Istruzione	Domain(istruzione)	Range(istruzione)
X ← Y + 2;	Y	X
Y ← X – 1;	X	Y

violano la condizione 2 e la condizione 3:
- range(istruzione A) ∩ domain(istruzione B) = X che è diverso dall'insieme vuoto ∅
- domain(istruzione A) ∩ range(istruzione B) = Y che è diverso dall'insieme vuoto ∅

3

Istruzione	Domain(istruzione)	Range(istruzione)
scrivi (X)	Y	∅
X ← X + Y + 3;	X, Y	X

violano la condizione 3:
- domain(istruzione A) ∩ range(istruzione B) = X che è diverso dall'insieme vuoto ∅

Se due (o più) istruzioni soddisfano le condizioni di Bernstein il risultato è indipendente dalla particolare sequenza di esecuzione eseguita dai processori (interleaving) e sarà quindi identico alla loro esecuzione seriale.

Quando anche una sola condizione viene violata si ottengono errori generati dal tempo dovuti al fenomeno dell'interferenza.

ESEMPIO 4 *Errore dovuto all'interleaving*

Un programma di magazzino utilizza due funzioni costituite da due istruzioni per aggiornare il totale di un prodotto a seconda che venga comprato o venga venduto.

```
procedura Scarica(x)
inizia
   TotS ← TANTI - x;
   TANTI ← TotS;
fine
```

```
procedura Carica(y)
inizia
   TotC ← TANTI + Y;
   TANTI ← TotC;
fine
```

Individuiamo per ogni coppia di istruzioni il *domain* e il *range* per verificare le condizioni di Bernstein:

domain(Scarica)= {TANTI, X, TotS}, range(Scarica)= {TANTI, TotS }
domain(Carica)= {TANTI, Y, TotC }, range(Carica)= {TANTI, TotC }

È immediato osservare che sono violate tutte e tre le condizioni e quindi molto probabilmente avremo errori dipendenti dal tempo.

Come verifica, supponiamo che contemporaneamente vengano vendute 3 unità e prodotte 5 unità a partire da una giacenza iniziale di 10 pezzi.

Analizziamo tre possibili sequenze di interleaving:

TANTI=10 TotS ← TANTI - 3; TotC ← TANTI + 5; TANTI ← TotC; TANTI ← TotS;	TANTI=10 TotS ← TANTI - 3; TANTI ← TotC; TotC ← TANTI + 5; TANTI ← TotS;	TANTI=10 TotC ← TANTI + 5; TotS ← TANTI - 3 TANTI ← TotS; TANTI ← TotC;
(TANTI =7)	(TANTI =12) ESECUZIONE SEQUENZIALE	(TANTI =15)

Solo nella seconda sequenza sono rispettati i vincoli temporali ed è quindi l'unica sequenza che genera il risultato esatto.

Nella seconda soluzione i due segmenti di codice sono stati eseguiti in sequenza, cioè non sono stati interrotti: quindi l'esecuzione in mutua esclusione delle istruzioni che modificano variabili condivise deve essere eseguita in modo tale da non essere interrotta (istruzioni atomiche).

Mutua esclusione e sezione critica

Consideriamo due processi in competizione per l'uso esclusivo di una risorsa comune: non è prevedibile sapere l'istante di tempo nel quale uno di essi utilizzerà la risorsa, ma bisogna garantire che quando ne entrerà in possesso lo farà in modo esclusivo, cioè la risorsa verrà utilizzata da "un processo alla volta" che la rilascerà al temine delle operazioni che la coinvolgono.

MUTUA ESCLUSIONE

Si ha mutua esclusione quando non più di un processo alla volta può accedere a una risorsa comune.

Le risorse comuni che utilizzeremo nei nostri programmi saranno delle variabili che verranno utilizzate dai diversi processi come buffer condiviso dove scrivere e leggere le informazioni che si devono scambiare.

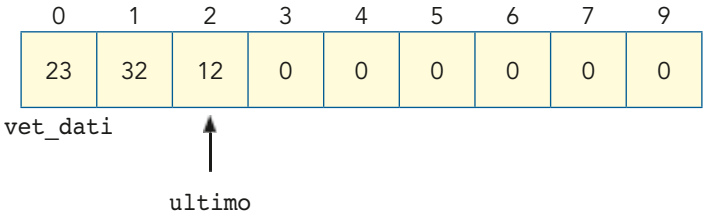
SEZIONE CRITICA

La sequenza di istruzioni con la quale un processo accede e modifica un insieme di variabili condivise prende il nome di *sezione critica*.

Vediamo un esempio nel quale due processi si scambiano dati attraverso la memoria condivisa per comprendere meglio perché è di fondamentale importanza regolare l'accesso a tale risorsa in modo esclusivo.

ESEMPIO 5 *Errore causato dalla interrompibilità delle operazioni*

Due processi P1 e P2 condividono una regione di memoria dove il produttore genera un numero e lo inserisce nella prima posizione libera di un vettore vet_dati[x] e il consumatore legge l'ultimo dato inserito azzerando il contenuto della cella.
Oltre che il vettore i processi condividono anche la variabile indice ultimo che contiene il valore dell'ultima cella occupata:



Riportiamo un segmento di codice per ciascuno di essi:

Produttore	Consumatore

funzione inserisci(nuovo_dato)
inizio
  ultimo ← ultimo+1
  vet_dati[ultimo] ← nuovo_dato
fine
	funzione preleva()
	inizio
	  letto r vet_dati[ultimo]
	  vet_dati[ultimo]<= 0
	  ultimo r ultimo -1
	fine

Entrambi i segmenti di codice accedono alle variabili condivise e modificano sia il contenuto del vettore che dell'indice: una esecuzione contemporanea potrebbe portare a situazioni errate, come nel caso seguente.
A causa dello scadere del **time slice** un processore potrebbe eseguire nel tempo le istruzioni in questa sequenza:

```
t0 Prod ultimo ← ultimo+1
t1 Cons letto ← vet_dati[ultimo]
t2 Cons ultimo ← ultimo -1
t3 Cons vet_dati[ultimo] ← 0
t4 Pros vet_dati[ultimo] ← nuovo_dato
```

Il consumatore andrebbe a prelevare un dato non ancora prodotto (quindi un valore uguale a zero) e aggiornerebbe l'indice facendo in modo che il produttore "sovrapponga" il nuovo dato a uno preesistente, non ancora prelevato.

Nel nostro esempio le sezioni critiche sono associate alle istruzioni che accedono alle variabili condivise vet_dati[x] e ultimo presenti nelle due funzioni descritte, inserisci() e preleva().

La regola di mutua esclusione stabilisce che in ogni istante una risorsa o è libera oppure è assegnata a uno e un solo processo: in particolare per avere la mutua esclusione devono essere soddisfatte le seguenti quattro condizioni:

- nessuna coppia di processi può trovarsi simultaneamente nella sezione critica;
- l'accesso alla regione critica non deve essere regolato da alcuna assunzione temporale o dal numero di CPU;
- nessun processo che sta eseguendo codice al di fuori della regione critica può bloccare un processo interessato a entrarvi;
- nessun processo deve attendere indefinitamente per poter accedere alla regione critica.

Starvation e deadlock

Un errata sincronizzazione può portare al fallimento delle elaborazioni, genera situazioni di incoerenza dei dati, e può portare a situazioni di blocco dei processi:

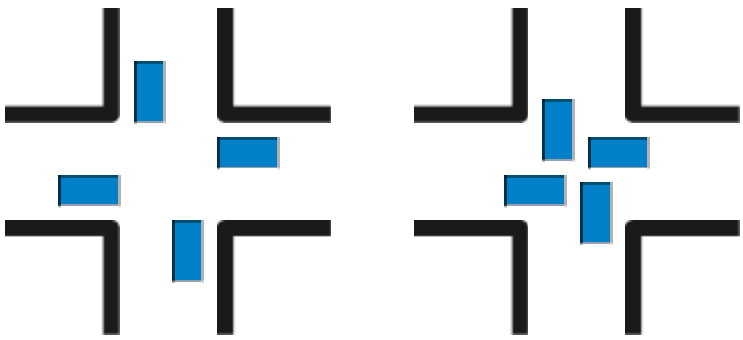
- starvation (o blocco individuale): si verifica quando un processo rimane in attesa di un evento che non accadrà mai, e quindi non può portare a termine il proprio lavoro,
- deadlock (stallo o blocco multiplo): si verifica quando due o più processi rimangono in attesa di eventi che non potranno mai verificarsi a causa di condizioni cicliche nel possesso e nella richiesta di risorse.

Vediamo un terzo esempio dove la mutua esclusione permette di risolvere il problema della interferenza ma può causare il blocco permanente dei processi.

ESEMPIO 6 *Blocco critico o deadlock*

Supponiamo di avere un incrocio con quattro strade, senza che questo venga regolato dalla presenza di un vigile e neppure di un semaforo.
La situazione di blocco si può verificare se contemporaneamente un automezzo giunge da ogni strada all'incrocio e tutti gli autisti si comportano allo stesso modo:

A danno la precedenza a destra, come previsto dal regolamento della strada;
B si apprestano ad attraversare l'incrocio senza curarsi degli altri automezzi.



Caso a: autisti diligenti Caso b: autisti incoscienti

In entrambe le situazioni si verifica un deadlock in quanto tutti gli automezzi rimangono bloccati senza possibilità di soluzione, cioè si ostacolano a vicenda creando la "morte" contemporanea e collettiva.

Diverso invece il caso nel quale una risorsa è occupata per un solo processo in modo continuativo: è il caso che capita quando siete in coda a uno sportello e continuano ad arrivare "furbi" che passano davanti, impedendovi di avanzare verso l'impiegato: in questo caso si tratta di starvation.

Possiamo riassumere in una tabella le diverse situazioni di interazione a seconda della natura dei processi:

Tipo	Relazione	Meccanismo	Problemi di controllo
Processi "ignari" uno dell'altro	Competizione	Sincronizzazione	Mutua esclusione Deadlock Starvation
Processi con conoscenza indiretta l'uno dell'altro	Cooperazione (sharing)	Sincronizzazione	Mutua esclusione Deadlock Starvation
Processi con conoscenza diretta l'uno dell'altro	Cooperazione (comunicazione)	Comunicazione	Deadlock Starvation

Proprietà

Un programma concorrente deve inoltre godere di alcune proprietà che devono essere valide per ogni possibile "storia di esecuzione" del programma stesso.

Abbiamo tre diverse proprietà:

- ◀ **safety** ▶: con questa proprietà si intende una condizione che deve sempre verificarsi per il buon fine dell'esecuzione del processo oppure una condizione di pericolo che non si deve mai verificare per la sicurezza del sistema: in altre parole, se un sistema avanza, va "nella direzione voluta" senza eseguire azioni indesiderate.
  I processi non devono "interferire" fra di loro nell'accesso alle risorse condivise e i meccanismi di sincronizzazione servono a garantire la proprietà di safety;
- fairness: è una proprietà connessa alla politica di scheduling del sistema operativo che deve sempre mandare in esecuzione qualsiasi processo, cioè tutte le richieste di esecuzione di un processo devono essere prima o poi soddisfatte; in questo modo viene garantita l'assenza di starvation, cioè si garantisce che tutti i processi "prima o poi" portino a compimento il proprio lavoro;
- ◀ **liveness** ▶: sta a indicare un evento che deve accadere in futuro, cioè deve garantire che il processo che avanza porterà a termine in modo corretto il proprio lavoro: i meccanismi di sincronizzazione devono fare in modo che un processo non aspetti "indeterminatamente" che una risorsa venga rilasciata oppure che *tutti* i processi si "*blocchino*" in attesa di eventi che non possono verificarsi: è la proprietà che esclude situazioni di deadlock.

◀ **Safety vs liveness** *Safety*: "nothing bad happens"; *liveness*: "something good eventually happens". ▶

Conclusioni

È sicuramente più complesso scrivere programmi concorrenti rispetto ai programmi sequenziali in quanto non basta essere sicuri della correttezza dei singoli moduli ma è necessario garantire il loro corretto funzionamento per ogni possibile combinazione di interazioni (volute o indesiderate) che questi possono avere.

Le condizioni di Bernstein ci permettono di individuare nel nostro codice dove possono verificarsi problemi dipendenti dal tempo, ma non sempre è semplice garantire la mutua esclusione e soprattutto effettuare il test e il debug di applicazioni che presentano race condition.
Nelle prossime lezioni descriveremo gli strumenti e le tecniche per scrivere programmi paralleli e garantire l'assenza di situazioni di starvation o di deadlock.

Esempio riepilogativo (non informatico)

A conclusione di questa lezione riportiamo un esempio non informatico che però racchiude i concetti fondamentali che sono stati esposti fino a ora, e dove possiamo facilmente comprendere gli effetti "disastrosi" che possono essere generati da errori dipendenti dal tempo o dalla errata sincronizzazione.

Situazione

Consideriamo un caso di parallelismo dal "mondo reale", dove una coppia di sposi conduce le proprie attività giornaliere:

- lei pianifica di riordinare la casa, andare a fare shopping oppure in palestra, passare a prendere due pizze e cenare a casa col marito;
- lui pianifica di lavorare mezza giornata, incontrare degli amici, portarli a casa nel pomeriggio per giocare a carte, uscire a comprare delle bevande e tornare a cenare con la moglie.

Analisi dei processi

In questo esempio si vedono in concreto alcuni concetti tipici della concorrenza:

- sincronizzazione: marito e moglie devono essere a cena assieme;
- processi indipendenti: marito e moglie possono svolgere gran parte delle attività che hanno pianificato in maniera del tutto indipendente;
- risorse condivise: la casa viene usata da entrambi gli attori in competizione fra loro;
- cooperazione: la cena sarà pronta per la cooperazione fra moglie (che acquista le pizze) e marito (che acquista da bere);
- non determinismo: la moglie non ha ancora deciso se fare shopping o palestra e a seconda della sua scelta alla fine della giornata il bilancio familiare sarà diverso.

È evidente come l'elemento cruciale è la risorsa condivisa e la dipendenza dal tempo dei singoli processi e il non determinismo può portare a situazioni indesiderate:

- **A** la moglie potrebbe prolungare le attività domestiche anche nel pomeriggio e quindi il marito troverebbe in casa la moglie quando vi torna con gli amici;
- **B** la moglie potrebbe rientrare prima del previsto avendo terminato in anticipo gli acquisti e rinunciando alla palestra e potrebbe trovare la casa "occupata".

Verifichiamo le conoscenze

Esercizi a scelta multipla

1 Come "correttezza temporale" si intendono due vincoli da soddisfare:

a) determinismo b) sincronizzazione c) timing constraint d) cooperazione

2 Indica quale non è una caratteristica degli errori dipendenti dal tempo:

a) indeterminati b) irriproducibili c) inconsistenti d) latenti

3 Quale tra le seguenti non è una condizione di Bernstein?

a) range(istruzione A) > range(istruzione B) = Ø
b) range(istruzione A) > domain(istruzione B) = Ø
c) domain(istruzione A) > range(istruzione B) = Ø
d) domain(istruzione A) ∩ domain (istruzione B) = Ø

4 Una errata sincronizzazione può portare:

a) all'interleaving b) all'overlapping c) alla starvation d) al deadlock

Test vero/falso

1	La programmazione concorrente introduce errori dipendenti dal tempo.	V F
2	La velocità della CPU è una delle cause degli errori dipendenti dal tempo.	V F
3	L'introduzione di ritardi risolve il problema degli errori dipendenti dal tempo.	V F
4	Nei sistemi real time i risultati devono essere prodotti entro la deadlines.	V F
5	Una risorsa allocata come risorsa dedicata necessita di sincronizzazione iniziale.	V F
6	Nei sistemi hard real time siamo in una situazione di real time stretto.	V F
7	L' interleaving si genera da una errata schedulazione dei processi.	V F
8	L'overlapping si genera da una errata schedulazione temporale dei processi.	V F
9	La violazione di una condizione di Bernstein crea l'interferenza e problemi legati al tempo.	V F
10	La proprietà di liveness esclude situazioni di deadlock.	V F

Domande a risposta aperta

1 Dai una definizione di mutua esclusione.
2 In che cosa consiste una regione critica?
3 In che cosa consiste l'interleaving?
4 In che cosa consiste l'overlapping?
5 Descrivi le condizioni di Bernstein.
6 In che cosa consiste la starvation?
7 In che cosa consiste il deadlock?
8 In che cosa consiste la proprietà di safety?
9 In che cosa consiste la proprietà di fairness?
10 In che cosa consiste la proprietà liveness?

LEZIONE 3

SINCRONIZZAZIONE TRA PROCESSI: SEMAFORI

IN QUESTA LEZIONE IMPAREREMO...

- a definire e utilizzare i semafori di basso livello e spin lock()
- il concetto di indivisibilità di una primitiva
- il funzionamento dei semafori di Dijkstra

Premessa: quando è necessario sincronizzare?

Nel caso di processi interagenti, siano essi in competizione, cioè che chiedono l'uso di una risorsa comune riusabile e di molteplicità finita per i propri scopi, oppure siano in cooperazione per raggiungere un obiettivo comune, possono verificarsi casi di interferenza.

La strategia da adottare per gestire l'interferenza è diversa per ogni situazione e dipende dalla tollerabilità che il sistema può permettersi degli effetti di una errata sincronizzazione, dalla possibilità di individuare agevolmente le eventuali situazioni e di poterle correggere ripristinando le situazioni precedenti ed eventualmente ripetendo le operazioni critiche.

Possiamo classificare in quattro gruppi la casistica di situazioni possibili e per ogni gruppo indicare l'azione che è necessario effettuare:

Conseguenze	Esempio	Strategie
inaccettabili	incrocio stradale	evitare ogni interferenza
trascurabili	applicazioni non critiche	ignorare
rilevabili e controllabili	iteratori	rilevare ed evitare
rilevabili e recuperabili	rete ethernet	rilevare e ripetere

Noi ci occuperemo del primo caso, cioè di situazioni in cui la competizione tra i processi ci porta a situazioni di interferenza che possono provocare situazioni inaccettabili e che quindi devono essere gestite con la mutua esclusione.

La regola di mutua esclusione stabilisce che sezioni critiche devono escludersi mutuamente nel tempo cioè che una sola sezione critica può essere in esecuzione a ogni istante.

La sezione critica viene gestita in modo che:
- **A** un processo che deve accedere a una regione critica deve chiedere l'autorizzazione eseguendo una serie di istruzioni che, nel caso la risorsa fosse libera, gli garantiscono il suo utilizzo esclusivo per tutta la durata della sua elaborazione; se la risorsa fosse occupata il gestore ne impedisce l'accesso gestendo la richiesta, per esempio, con una coda di attesa;
- **B** quando un processo termina di utilizzare una risorsa, ha quindi terminato l'esecuzione delle istruzioni della sezione critica, deve effettuare un insieme di operazioni per rilasciarla in modo che possa essere utilizzata dagli altri processi.

Mediante l'utilizzo di primitive che regolino l'accesso e il rilascio della risorsa è necessario garantire che:
- **A** la risorsa o è libera oppure è utilizzata da un solo processo (condizione di mutua esclusione);
- **B** i processi devono sempre poter accedere alla risorsa richiesta e portare a termine il proprio lavoro (condizione di fairness).
- **C** i processi non devono avere cicli di ritardo non necessari che possano rallentare l'accesso alla regione critica da parte di un altro processo.

> Dobbiamo scrivere i nostri programmi in modo da garantire la serializzazione dell'uso della risorsa e l'utilizzo della stessa per un tempo finito, in modo che tutti i processi che ne hanno bisogno prima o poi la possano utilizzare.

Semafori di basso livello e spin lock()

Affronteremo la trattazione del caso di richiesta di risorsa singola da parte di soli due processi ma il concetto è immediatamente estendibile a un numero qualsivoglia di processi: più complesso è invece il caso di risorse multiple richieste contemporaneamente dagli stessi processi, che discuteremo nella prossima lezione.

Il primo meccanismo che analizziamo è quello che associa a ogni risorsa una variabile x che in base al suo valore assume il seguente significato:

x = 1 risorsa libera, cioè nessun processo la sta utilizzando;
x = 0 risorsa occupata da un processo.

Quindi il flag fa la funzione di un semaforo:

x = 1 semaforo verde, è possibile accedere alla risorsa;
x = 0 semaforo rosso, la risorsa è occupata ed è necessario mettersi in attesa che si liberi.

> La variabile x semaforo può assumere solamente il valore 0 oppure 1: l'implementazione di questi semafori può essere realizzata con variabili booleane che prendono il nome di spinlock.

Vediamo come esprimere in pseudocodifica i segmenti di codice per effettuare rispettivamente l'allocazione e il rilascio di una risorsa.

Allocazione di una risorsa: lock()

La primitiva (o funzione) che permette di allocare una risorsa prende il nome di lock().
Possiamo indicare la sua sintassi nel seguente formato:

```
lock(x);
```

dove x è il semaforo associato alla risorsa che desideriamo utilizzare.

La primitiva lock() deve:
- testare il semaforo per verificarne il suo colore;
- se è verde, modificarne il valore a rosso;
- se è rosso, aspettare che diventi verde per poi metterlo a rosso.

In pseudocodifica una possibile realizzazione è la seguente:

```
funzione lock(x)
inizia
   ripeti          // ciclo di attesa sul semaforo nel caso che sia rosso
   finche x=1      // esci dal ciclo a semaforo verde
   x ← 0;          // metti il semaforo a rosso
fine
```

L'osservazione che possiamo fare a questa funzione è che il ciclo di attesa viene eseguito ripetendo continuamente il test sul semaforo fino a quando diventa verde (x=1): durante la fase di attesa un processo "consuma inutilmente CPU" e questo tipo di situazione prende il nome di attesa attiva.

Rilascio di una risorsa: unlock()

La primitiva (o funzione) che permette di rilasciare una risorsa prende il nome di unlock(). Possiamo indicare la sua sintassi nel seguente formato:

```
unlock(x);
```

dove x è il semaforo associato alla risorsa che desideriamo utilizzare.

La primitiva unlock() deve semplicemente modificare il valore del semaforo da rosso a verde, quindi:

```
funzione unlock(x)
inizia
   x ← 1;          // metti il semaforo a verde
fine
```

La mutua esclusione si ottiene facendo precedere la lock(x) a una sezione critica e facendola seguire da una unlock().

```
…
lock(x);
< sezione critica>
unlock(x);
…
```

Problema della indivisibilità

Per come abbiamo scritto l'istruzione di lock() non possiamo garantire l'acceso seriale a una risorsa, in quanto potrebbe verificarsi una situazione di interleaving indesiderata.

Vediamo per esempio la seguente situazione:

1. ipotizziamo che il semaforo sia verde x = 1
2. Il processo P1 effettua la lock(x) fino a verificare la condizione di test ma viene sospeso prima che ne possa modificare il valore;
3. un secondo processo P2 effettua anch'esso la lock(x) e, trovando il semaforo verde, lo mette a rosso e inizia a utilizzare la risorsa;
4. quando viene risvegliato il processo P1 esegue l'istruzione che pone a rosso il semaforo (che di fatto però è già rosso) e anch'esso utilizza la risorsa.

Risulta perciò violata la mutua esclusione in quanto i due processi si trovano contemporaneamente a utilizzare la risorsa.

Per evitare questa situazione è necessario rendere indivisibile l'esecuzione delle istruzioni della funzione lock(x): la soluzione potrebbe essere quella di disabilitare le interruzioni all'inizio e al termine di questa funzione in modo che diventi ininterrompibile.

Per esempio il codice potrebbe essere il seguente:

```
funzione lock(x)
inizia
<disabilitare le interruzioni>
   ripeti                // ciclo di attesa sul semaforo nel caso che sia rosso
   finche x=1            // esci dal ciclo a semaforo verde
   x<-0;                 // metti il semaforo a rosso
<abilitare le interruzioni>
fine
```

Analogo problema potrebbe verificarsi per interleaving anche sulla primitiva di unlock(x) e quindi anch'essa viene eseguita a interruzioni disabilitate.

```
funzione unlock(x)
inizia
<disabilitare le interruzioni>
   x<-1;                 // metti il semaforo a verde
<abilitare le interruzioni>
fine
```

Continuare ad abilitare e disabilitare le interruzioni porta però a compromettere le prestazioni del sistema: se osserviamo attentamente le due primitive ci accorgiamo che il problema si verifica a causa dell'interruzione tra il test del semaforo e il suo settaggio al nuovo valore.
Introduciamo a livello macchina (quindi hardware) una nuova istruzione, che chiamiamo TestAndSet(x), che controlla e modifica il valore di un bit in modo ininterrompibile, e una funzione Set(x) che ne effettua il semplice settaggio, da utilizzarsi nella unlock().

Riscriviamo le due primitive utilizzando queste due nuove istruzioni:

```
funzione lock(x)
inizia
```

```
   ripeti                // ciclo di attesa sul semaforo nel caso che sia rosso
   <ritardo>
   finche TestAndSet(x)  // esci dal ciclo settando il semaforo a rosso
fine
```

e

```
funzione unlock(x)
inizia
   set(x);               // metti il semaforo a verde
fine
```

Nonostante questo miglioramento rimane da risolvere il problema della attesa attiva; inoltre non abbiamo la garanzia esplicita che un processo non attenda indefinitamente su di un semaforo che, anche se diviene verde, viene ripetutamente assegnato ad altri processi e non a lui.

Gli spinlock non vengono quindi utilizzati come meccanismo di sincronizzazione ma sono alla base della realizzazione di primitive più complesse.

Zoom su...

TSL

I primi a notare la necessità di avere una istruzione indivisibile furono i progettisti dell'OS/360 che aggiunsero nel linguaggio macchina del sistema l'istruzione TEST AND SET LOCK (TSL).

Semafori di Dijkstra

E.W. Dijkstra nel 1968 ha proposto due primitive che permettono la soluzione di qualsiasi problema di interazione fra processi, che sono:

- la primitiva P(S), che riceve in ingresso un numero intero S non negativo (semaforo), che viene utilizzata per accedere alla risorsa;
- la primitiva V(S), che riceve anch'essa in ingresso un numero intero S non negativo (semaforo), che viene utilizzata per rilasciare la risorsa.

Introduciamo quindi un nuovo tipo di dato, il semaphore, dove una sua istanza non è altro che una variabile intera non negativa alla quale è possibile accedere solo tramite le due primitive P(S) e V(S).

ESEMPIO 7 *Il primo semaforo*

Dichiarazione di un oggetto di tipo semaphore:

```
semaphore s = vi;
```

dove vi (vi≥0) è il *valore iniziale* e il valore 0 corrisponde al rosso.

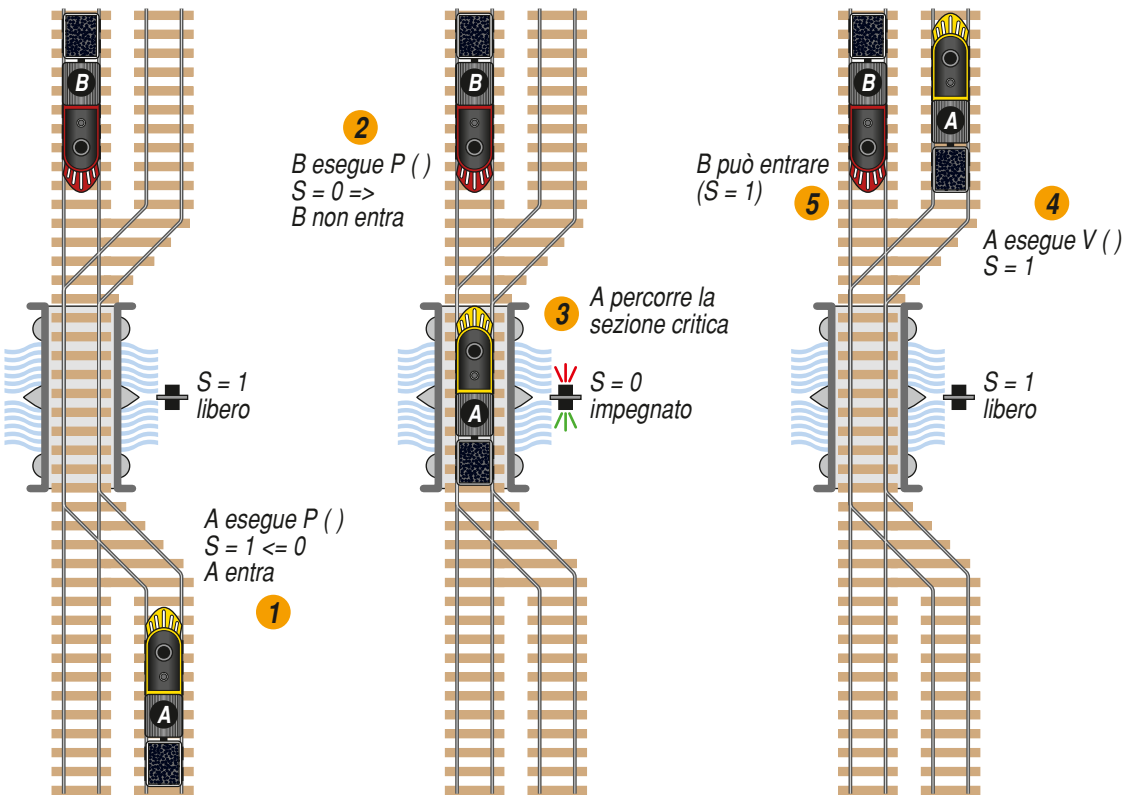
L'idea di Dijkstra è quella di disciplinare l'accesso alle risorse mediante code e rendendo inattive le situazioni di attesa: se un semaforo è rosso il processo che fa il test (richiama la primitiva P(S)) si sospende e viene messo nella coda dei processi, in attesa che si liberi quella risorsa mentre il processo che rilascia la risorsa invoca la primitiva V(S) che modifica il valore del semaforo S e risveglia il primo processo presente nella coda di attesa.

Quindi due o più processi possono cooperare attraverso semplici segnali, in modo tale che un processo possa essere bloccato in specifici punti del suo programma finché non riceve un segnale da un altro processo.

ESEMPIO 8 *Semafori ferroviari*

Il nome "semaforo" proviene dalla segnalazione ferroviaria e facciamo un esempio di gestione rimanendo in ambito ferroviario: supponiamo di avere la situazione rappresentata in figura, dove una linea a due binari a un certo punto deve confluire su di un ponte con binario unico.

Il ponte è la risorsa condivisa che dovrà essere gestita in mutua esclusione.

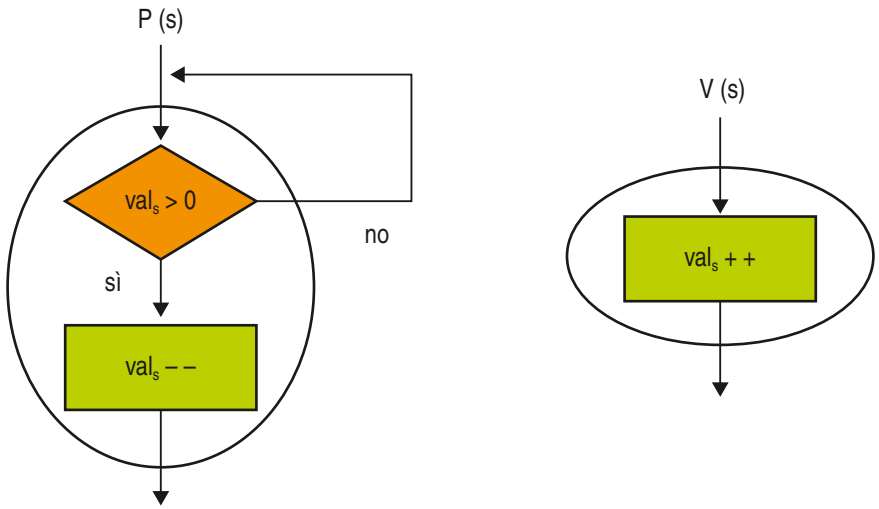


Associamo al ponte un semaforo che viene controllato e gestito dai macchinisti dei treni: quando un treno A si avvicina al ponte osserva il semaforo.

1. Il macchinista di A controlla lo stato del semaforo (esegue una P(S)): lo trova spento, quindi lo accende (S = 0) e inizia a transitare sul ponte;

2. anche il treno B sopraggiunge al ponte ma trovando il semaforo acceso esegue anch'esso una P(S), si ferma e rimane in attesa.
3. A continua la sua corsa attraversando il ponte;
4. una volta raggiunta l'altra sponda, spegne il semaforo (S=1);
5. B ora vede il semaforo spento e lo accende (S=0) e inizia pure lui ad attraversare il ponte.

Lo schema a blocchi delle due istruzioni è il seguente:



La variabile S non è un singolo bit ma un numero intero e, come vedremo, le primitive P(S) e V(S) permettono di regolare l'acceso multiplo e non solo di due processi.

La pseudocodifica delle due primitive è la seguente:

```
funzione P(S)
inizia
 se S=0            // risorsa occupata - semaforo rosso
   allora
    <il processo viene posto nella coda di attesa>
   altrimenti
    S ← S-1;       // accedi alla risorsa
fine
```

e

```
funzione V(S)
inizia
 se <è presente un processo in attesa>
   allora
    <poni il primo processo nello stato di pronto>
   S ← S+1;        // rilascia la risorsa
fine
```

Zoom su...

ORIGINE DI P E V

Originariamente queste primitive avevano il nome di wait e signal: Dijkstra li sostituì con le iniziali P e V di due termini olandesi:

- V è l'iniziale dal termine olandese verhogen: aumentare, alzare di livello;
- P è l'iniziale dal termine olandese proberen: provare, testare.

Un processo che esegue una P(S) può avanzare solo se trova S > 0, altrimenti deve accodarsi per attendere passivamente una V(S); un processo che esegue una V(S), se non esistono processi in attesa dentro la coda, ha come unico effetto quello di incrementare S, altrimenti risveglia uno dei processi che attendono nella coda: successivamente, in entrambe le situazioni, continua la propria evoluzione.

Naturalmente le primitive P(S) e V(S) devono essere indivisibili per evitare la sequenza di interleaving indesiderata e la procedura di assegnazione della risorsa ai processi in attesa viene stabilita in modo tale da garantire la proprietà di fairness, per esempio con una coda FIFO.

In un sistema uniprocessore si realizzano introducendo spinlock e/o disabilitando/riabilitando gli interrupt all'inizio/fine di ciascuna di esse (dato che sono implementate direttamente dal sistema operativo e l'intervallo temporale in cui gli interrupt sono disabilitati è molto breve).

La realizzazione completa delle due primitive è la seguente:

```
funzione P(S)
inizia
 <disabilita le interruzioni>
 LOCK(SX)
 se S=0
   allora
      <il processo viene posto nella coda di attesa>
   altrimenti
      S ← S-1;          // accedi alla risorsa
 UNLOCK(SX)
 <abilita le interruzioni>
fine
```

```
funzione V(S)
inizia
 <disabilita le interruzioni>
 LOCK(SX)
 se <è presente un processo in attesa>
   allora
      <poni il primo processo nello stato di pronto>
```

```
 S ← S+1;                // rilascia la risorsa
 UNLOCK(SX)
 <abilita le interruzioni>
fine
```

Si può verificare lo spin lock SX utilizzato per garantire l'indivisibilità delle due primitive: è necessario solamente nei sistemi multi-processore mentre per i sistemi uniprocessore è sufficiente disabilitare le interruzioni.

Semafori binari vs semafori di Dijkstra

I semafori di Dijkstra vengono anche chiamati **semafori generalizzati** (o a conteggio) per distinguerli dai semafori binari.
In letteratura si trovano spesso i nomi down(S) e up(S) per indicare rispettivamente P(S) e V(S) dato che l'istruzione P(S) decrementa di 1 il valore del semaforo (down(S)) e l'istruzione V(S) invece la incrementa (up(S)).

Naturalmente un semaforo di Dijkstra può essere utilizzato come semplice semaforo binario utilizzando solamente i valori 0 e 1.

Molteplicità di una risorsa

I semafori a conteggio vengono utilizzati per controllare l'accesso a una risorsa disponibile in un numero finito di esemplari: noi vedremo nel seguito una applicazione nel caso di utilizzo di un array di NUM celle come risorsa condivisa, ciascuna di esse destinata a contenere una variabile da condividere e quindi il semaforo verrà inizializzato al valore NUM per indicare che sono disponibili NUM risorse e quindi il semaforo rimane verde finché non sono state tutte "occupate".

Al test sul semaforo S possiamo avere due situazioni:
1. S = X dove x<=NUM è il numero di esemplari di risorsa liberi: se X > 0 il processo può accedere come in presenza di semaforo verde;
2. S = 0 risorse occupate, cioè 0 risorse libere e quindi il semaforo è rosso.

Le istruzioni di P(S) e V(S) incrementano e decrementano la molteplicità di risorsa libera:
- quando viene effettuata una P(S) su semaforo che ha valore >0 ne viene decrementato di 1 il suo valore, dato che viene occupata "una sua parte";
- quando viene effettuata una V(S) su un semaforo viene incrementato di una unità il suo valore dato che ne viene resa disponibile "una parte" per gli altri processi.

Verifichiamo le conoscenze

1 **Indica l'azione che è necessario effettuare per ciascuno dei quattro gruppi di conseguenze a fronte di situazioni di interferenza:**

Conseguenze	Strategie
inaccettabili	
trascurabili	
rilevabili e controllabili	
rilevabili e recuperabili	

Esercizi a scelta multipla

1 **In uno spin lock, a seconda del valore di x abbiamo:**

a) x = 1 risorsa libera
b) x = 0 risorsa libera
c) x = 1 risorsa occupata
d) x = 0 risorsa occupata

2 **L'istruzione TestAndSet(x):**

a) modifica il valore di un bit in modo ininterrompibile
b) modifica il valore di un byte in modo ininterrompibile
c) modifica il valore di x in modo ininterrompibile
d) tutte le affermazioni precedenti sono valide: dipende dalla applicazione

3 **L'istruzione TestAndSet(x):**

a) risolve il problema della attesa attiva
b) risolve il problema della starvation
c) risolve il problema dell'interleaving
d) non risolve alcun problema

4 **Indica con lo stesso numero le primitive che eseguono la stessa funzione:**

a) ..... P(S)
b) ..... V(S)
c) ..... WAIT(S)
d) ..... SIGNAL(S)
e) ..... UP(S)
f) ..... DOWN(S)

Esercizi a scelta multipla

1 Se la mancanza di sincronizzazione provoca danni trascurabili la miglior strategia è quella di ignorare il problema. V F

2 La serializzazione può essere ottenuta introducendo cicli di ritardo. V F

3 L'istruzione di lock su uno spin può generare una attesa attiva. V F

4 Per evitare il verificarsi di interleaving la primitiva look(x) si esegue a interruzioni disabilitate. V F

5 La primitiva di unlook(x), essendo una sola istruzione, non richiede interruzioni disabilitate. V F

6 Dijkstra disciplina l'accesso alle risorse mediante code e rendendo inattive le situazioni di attesa. V F

7 Nel semaforo di Dijkstra il valore iniziale 0 corrisponde al rosso. V F

8 Le primitive P(S) e V(S) non necessitano di essere indivisibili in quanto sono atomiche. V F

9 La P(S) su un semaforo rosso sospende il processo e lo mette in una coda. V F

10 La V(S) su un semaforo verde mette il semaforo a rosso senza sospendere il processo. V F

Domande a risposta aperta

1 Commenta e completa i seguenti segmenti di codice.

```
funzione lock(x)          // .....
inizia
<disabilitare le interruzioni>
   ripeti                 // .....
   finche x=1             // .....
   x ← 0;                 // .....
<abilitare le interruzioni>
fine
```

```
funzione unlock(x)        // .....
inizia
<disabilitare le interruzioni>
   x ← 1;                 // .....
<abilitare le interruzioni>
fine
```

```
funzione P(S)             // .....
inizia
<.....>
LOCK(SX)
   se S=0
      allora
         <.....>
      altrimenti
         S ← S-1;         // .....
UNLOCK(SX)
<.....>
fine
```

```
funzione V(S)
inizia
<.....>
LOCK(SX)
   se <.....>
      allora
         <.....>
   S ← S+1;               // .....
UNLOCK(SX)
<.....>
fine
```

LEZIONE 4

APPLICAZIONE DEI SEMAFORI

IN QUESTA LEZIONE IMPAREREMO...

- a realizzare la mutua esclusione mediante i semafori
- a regolare l'acceso multiplo tramite i semafori
- a utilizzare i semafori per realizzare i vincoli di precedenza

Semafori e mutua esclusione

Il semaforo viene utilizzato come strumento di sincronizzazione tra processi concorrenti che intendono cooperare, come per esempio nelle situazioni produttore-consumatore; per garantire la mutua esclusione nel caso di un singolo produttore e un singolo consumatore è sufficiente utilizzare semafori binari, cioè associare alla variabile il valore 1 per libero e 0 per occupato (come per gli spinlock).

Nella letteratura l'istanza di un semaforo viene generalmente indicata con l'identificatore mutex che deriva dalla contrazione dell'inglese mutual exclusion (mutua esclusione), e anche nella nostra trattazione utilizzeremo questa terminologia.

Scriviamo lo schema del codice di due processi che accedono concorrentemente a una risorsa condivisa:

```
programma concorrente MutuaEsclusione
semaphore mutex = 1;
. . .
Processo operazione1()
inizio
…
P(mutex);                          /* prologo */
<corpo della funzione produci>
V(mutex);                          /* epilogo */
…
fine
```

```
Processo operazione2 ()
...
inizio
P(mutex);                          /* prologo */
<corpo della funzione consuma>
V(mutex);                          /* epilogo */
…
fine

inizia /* principale*/
…
cobegin
   operazione1 ()
   operazione2 ()
coend
…
fine
```

È possibile dimostrare che la soluzione proposta risolve correttamente il problema della mutua esclusione, in quanto soddisfa le condizioni necessarie:
- le sezioni critiche devono essere eseguite in modo mutuamente esclusivo;
- non si devono verificare situazioni in cui i processi impediscono mutuamente la prosecuzione della loro esecuzione (deadlock);
- quando un processo si trova all'esterno di una sezione critica non può rendere impossibile l'accesso alla stessa sezione ad altri processi.

ESEMPIO 9 *Prenotazione posti al cinema*

In una sala cinematografica con più casse e un sistema di prenotazione via Internet per la vendita dei biglietti, il numero di posti è continuamente aggiornato e memorizzato in una variabile intera. Realizziamo la procedura che aggiorna il numero di posti ancora disponibili.
Introduciamo la variabile condivisa `postiLiberi` inizializzata al numero massimo di spettatori (per esempio 200).
Le prenotazioni sono effettuate richiedendo un numero quanti di posti maggiore di 0: se è possibile soddisfare la richiesta, i posti vengono occupati altrimenti viene data una segnalazione visiva.
La regione critica viene gestita con un semaforo che garantisce la mutua esclusione alla risorsa condivisa (`postiLiberi`).

Una possibile pseudocodifica è la seguente:

```
programma concorrente MutuaEsclusione
semaphore mutex = 1;
int postiLiberi=200;
. . .
Processo occupa(int quanti)
inizio
   P(mutex);                       /*prologo*/
      se postiLiberi>quanti
         allora
```

```
            postiLiberi= postiLiberi-quanti
         altrimenti
            scrivi("posti non disponibili)"
   V(mutex);                        /*epilogo*/
fine
```

Mutua esclusione tra gruppi di processi

In alcuni casi è consentito a più processi di eseguire contemporaneamente la stessa operazione su una risorsa, ma non alternare operazioni diverse: più processi possono cioè effettuare la medesima operazione contemporaneamente ma, per eseguire una operazione diversa da quella che stanno eseguendo tutti gli altri, è necessario che tutti questi finiscano di utilizzarla e che quindi la risorsa risulti libera da ogni processo; solo allora è possibile che un processo esegua una nuova operazione e se altri processi desiderano svolgere la medesima operazione la possono fare contemporaneamente al primo.

Indichiamo con `operazioneK` il prototipo della generica operazione che ha la seguente struttura:

```
procedura operazioneK ()
inizio
   <operazioni preliminari>         // prologo
   <corpo della funzione>
   <operazioni conclusive>          // epilogo
fine
```

Le operazioni preliminari che il processo deve eseguire (prologo) sono quelle inerenti al controllo degli accessi, cioè il processo che ha chiamato l'operazione `operazioneK` si deve sospendere se sulla risorsa sono in esecuzione operazioni diverse previste nel codice di `operazioneK`; nelle altre situazioni, cioè quando la risorsa è libera oppure utilizzata da un altro processo che fa la medesima operazione, si deve consentire al processo di accedere alla risorsa.
Le operazioni conclusive nel caso in cui il processo che sta uscendo è l'ultimo (o il solo) presente nella regione critica consistono nel liberare la risorsa per le altre attività, altrimenti si deve semplicemente aggiornare il numero dei processi presenti.

> Nelle istruzioni del prologo e dell'epilogo è quindi necessario utilizzare una *variabile di conteggio* dei processi presenti contemporaneamente nella regione critica, che chiameremo `ProcessiDentro` e anch'essa, dato che è in condivisione con gli altri processi, viene aggiornata all'interno di sezioni critiche che dovranno essere gestite in mutua esclusione: si introduce un nuovo semaforo che viene utilizzato appositamente per regolare l'accesso dei processi che devono modificare il contatore.

Scriviamo un primo affinamento della procedura, introducendo la nuova variabile `mutexPDK` che è il semaforo che regola l'aggiornamento della variabile `ProcessiDentroK` della `operazioneK`;

```
semaphore mutex =1, mutexPDK = 1;
int ProcessiDentroK = 0;
procedura operazioneK ( )
inizia
   // prologo
```

```
   P(mutexPDK)
      <controllo e aggiornamento ProcessiDentroK >
   V(mutexPDK)
   / parte centrale
   <corpo della funzione operazioneK >
   // epilogo
   P(mutexPDK)
      <controllo e aggiornamento ProcessiDentroK >
   V(mutexPDK)
fine
```

Le operazioni effettuate dal prologo sono di seguito elencate e commentate:

```
P(mutexPDK)                          //accesso in mutua esclusione alla RC
   ProcessiDentroK ++;               //aggiorna numero processi
   if (ProcessiDentroK ==1)          //se è il primo processo che accede
      P(mutex)                       //mette a rosso il semaforo sulla risorsa
V(mutexPDK)                          //libera accesso in mutua esclusione alla RC
```

Le operazioni effettuate dall'epilogo sono di seguito elencate e commentate:

```
P(mutexPDK)                          //accesso in mutua esclusione alla RC
   ProcessiDentroK --                //aggiorna numero processi
   if (ProcessiDentroK ==0)          //se l'ultimo che utilizza la risorsa
      V(mutex)                       //mette a verde il semaforo sulla risorsa
V(mutexPDK)                          //libera accesso in mutua esclusione alla RC
```

Il codice completo è il seguente:

```
semaphore mutex =1, mutexPDK = 1;
int ProcessiDentroK = 0;
procedura operazioneK( )
inizia
// prologo
   P(mutexPDK)
      //controllo e aggiornamento ProcessiDentroK
      ProcessiDentroK ++;            //aggiorna numero processi
      if (ProcessiDentroK ==1)       //se è il primo processo che accede
         P(mutex)                    //mette a rosso il semaforo sulla risorsa
   V(mutexPDK)
// parte centrale
   < corpo della funzione operazioneK >
// epilogo
   P(mutexPDK)
      // controllo e aggiornamento ProcessiDentroK
      ProcessiDentroK --             //aggiorna numero processi
      if (ProcessiDentroK ==0)       //se l'ultimo che utilizza la risorsa
         V(mutex)                    //mette a verde il semaforo sulla risorsa
   V(mutexPDK)
fine
```

È immediata l'estensione nel caso di un insieme di attività simili da eseguirsi contemporaneamente: basta scrivere una procedura per ogni operazione e aggiungere un contatore e un semaforo riservato per regolarne l'accesso a ciascuna di esse.
Deve invece rimanere unico il semaforo mutex che regola l'accesso alla risorsa comune elaborata all'interno del <corpo della funzione operazioneK>.

ESEMPIO 10 ***Tre semafori***

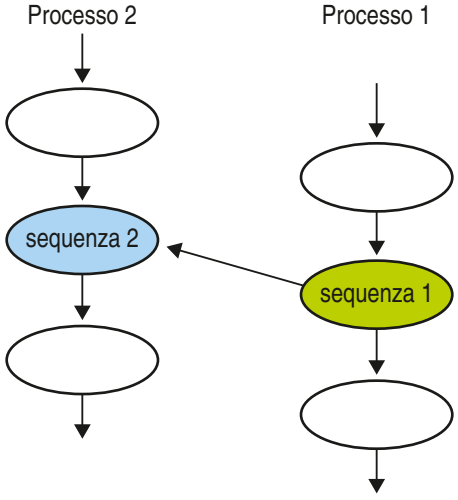
Uno schema per il caso di tre operazioni è il seguente:

```
semaphore mutex =1                  // comune a tutte le procedure
semaphore mutexPD1=1, mutexPD2=1, mutexPD3=1; // uno per procedura
int ProcessiDentro1 = 0;            // uno per procedura
int ProcessiDentro2 = 0;
int ProcessiDentro3 = 0;
procedura operazione1()
. . .
fine
procedura operazione2()
. . .
fine
procedura operazione3()
. . .
fine
```

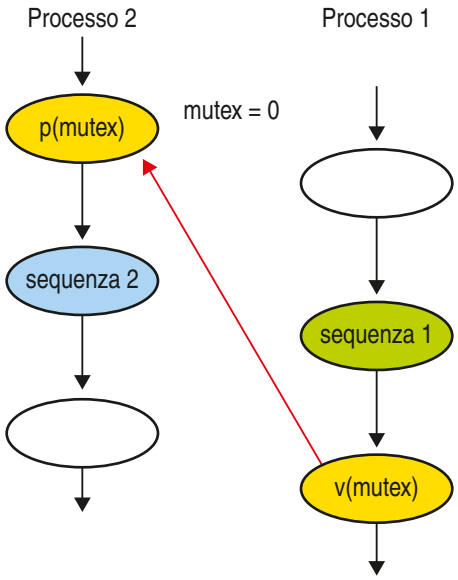
Semafori come vincoli di precedenza

I semafori binari possono anche essere utilizzati per stabilire dei vincoli di precedenza sull'esecuzione di gruppi di operazioni in processi paralleli.

Supponiamo di avere due sequenze di istruzioni sequenza1 e sequenza2 che devono essere eseguite da due processi diversi necessariamente in successione, cioè il secondo processo esegue la sequenza2 solo dopo che il primo processo ha eseguito la sequenza1.

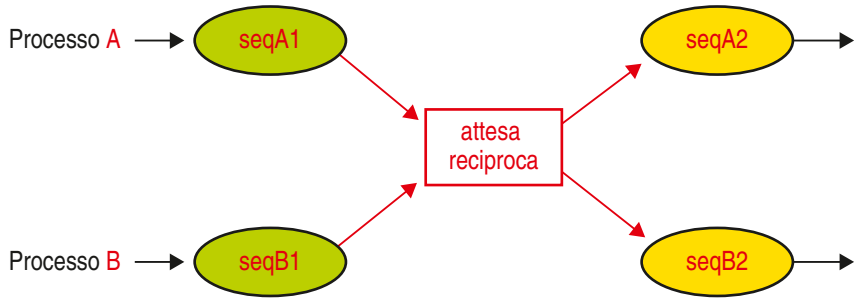


Introduciamo un semaforo mutex che inizializziamo a rosso (mutex=0) e che viene messo a verde dal processo1 solo dopo che ha eseguito la sequenza1: il secondo processo, prima di eseguire la sequenza2, effettua un test su questo semaforo che troverà *verde* solo dopo l'esecuzione della sequenza2 altrimenti, trovandolo *rosso*, aspetta il processo1.



Problema del rendez-vous

I semafori binari possono anche essere utilizzati per gestire dei "rendez-vous" tra due processi.
Per esempio, supponiamo che due processi debbano eseguire due sequenze per ciascuno ma per ogni processo la seconda sequenza deve essere eseguita dopo che anche l'altro processo ha eseguito la prima: in altre parole il processo più veloce deve attendere il processo più lento.

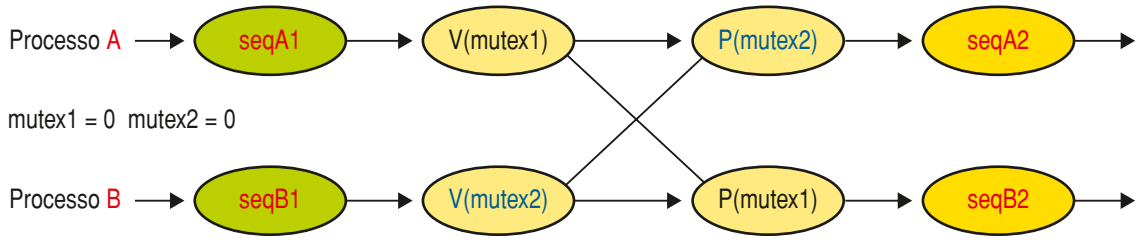


La sequenza seqA2 deve essere eseguita dopo la sequenza seqA1 e la sequenza seqB1 così pure la sequenza seqB2 deve essere eseguita dopo le sequenza seqA1 e seqB1.

Introduciamo due semafori in modo che i due processi si possano scambiare "segnali temporali in modo simmetrico": quando un processo giunge "all'appuntamento" segnala all'altro di esserci arrivato e lo attende.

Poniamo all'inizio dell'esecuzione i due semafori a rosso e facciamo in modo che ogni processo metta a verde il semaforo che ferma la prosecuzione dell'elaborazione dell'altro processo e succes-

sivamente testa il semaforo che è gestito dall'altro processo che gli permette di proseguire, come riportato nel seguente diagramma:



Se arriva prima il processoA al rendez-vous, dopo aver messo a verde il mutex1 trova mutex2 rosso e quindi si ferma fino a che sopraggiunge il processoB a mettere a verde il mutex2, risvegliando il processoA che riprende la sua evoluzione: il processoB può proseguire perché trova a verde il semaforo mutex1 settato dal processoA come prima operazione al suo arrivo alla zona di sincronizzazione, quindi ora entrambi possono procedere eseguendo rispettivamente la seqA2 e la seqB2.

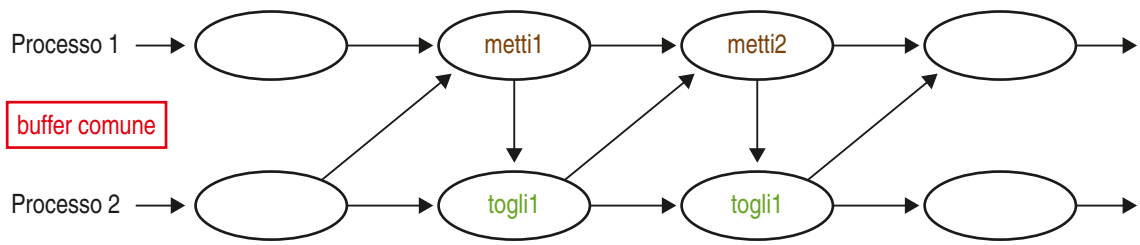
ESEMPIO 11 *Rendez-vous prolungato*

Scriviamo un esempio dove due processi, per poter proseguire, devono scambiarsi alcuni dati utilizzando un buffer condiviso: il primo processo scrive un dato alla volta, ma per poter scrivere il secondo dato deve attendere che il secondo processo prelevi il primo, e così via.

Individuiamo i vincoli di sincronizzazione:
- i processi devono effettuare l'accesso al buffer comune in mutua esclusione;
- il processo P2 può prelevare un dato solo dopo che P1 lo ha inserito;
- il processo P1, prima di inserire un nuovo dato, deve attendere che P2 abbia estratto il precedente.

> Graficamente indichiamo solamente che devono essere scambiati due dati, ma il programma che scriviamo non ha limiti di funzionamento sul numero di dati da scrivere e leggere.

Graficamente la situazione è la seguente



La realizzazione della sincronizzazione prevede l'utilizzo di due semafori:
- mutexBV: per realizzare l'attesa del processo1 in caso di buffer pieno;
- mutexBP: per realizzare l'attesa del processo2 in caso di buffer vuoto.

Inizialmente il buffer è vuoto e i due semafori sono così settati:

```
buffer = <vuoto>
mutexBV= 1 //all'inizio il semaforo che indica buffer vuoto è verde
mutexBP= 0 //all'inizio il semaforo che indica buffer pieno è rosso
```

Il processo che scrive effettua il test sul semaforo mutexBV che indica se il buffer è vuoto, quindi se lo trova verde lo mette a rosso, riempie il buffer e setta a verde il semaforo mutexBP indicando che ora il buffer è pieno, pronto per la lettura:

```
Procedura invio(dato)
inizio
...
P(mutexBV)                              // se il buffer è vuoto riempilo
inserisci(dato)
V(mutexBP)                              // indica che il buffer è pieno
...
fine
```

Il processo che deve leggere il contenuto del buffer per prima cosa testa il semaforo mutexBP che indica se il buffer è pieno, quindi lo svuota e successivamente setta a verde il semaforo mutexBV indicando che il buffer è vuoto, pronto per una nuova scrittura:

```
Procedura ricezione( )
inizio
...
P(mutexBP)                              // se il buffer è pieno svuotalo
estrai(dato)
V(mutexBV)                              // indica che il buffer è vuoto
...
fine
```

Verifichiamo le competenze

1 Considera i seguenti processi,

```
Risorse condivise
   semaphore S=1;
   int x=0;

Processo P1{
   while (true) do
   begin
      down(S);
      x:=x+1;
      up(S);
   end
}
```

```
Processo P2{
   while (true) do
   begin
      down(S);
      write(x);
      up(S);
   end
}
```

e supponi che i processi vengano eseguiti concorrentemente sulla stessa CPU.
Quali sono le sezioni critiche in questo programma?
Vale la proprietà di mutua esclusione?
Descrivi il comportamento e i possibili output del programma concorrente.

2 Scrivi lo pseudocodice dove due processi devono scambiarsi reciprocamente dei dati per poter proseguire utilizzando un buffer condiviso: il primo processo che scrive un dato prima di proseguire deve attendere che il secondo processo lo prelevi e inserisca a sua volta un dato destinato a lui.

3 Scrivi lo pseudocodice di tre processi concorrenti che risolvono il seguente problema:
si vogliono sommare gli elementi di un array 'buf' di 20 elementi, cioè fare l'operazione buf[0]+buf[1]+…+buf[19]. L'operazione viene fatta però in modo concorrente, in modo tale cioè che il primo thread sommi gli elementi di posizione pari dell'array, mentre il secondo sommi gli elementi di posizione dispari. Il terzo thread somma i risultati dei primi due.
Quanti semafori sono necessari? Come devono essere inizializzati i semafori?
Scrivi una pseudocodifica della soluzione.

4 Un conto corrente richiede che alla variabile saldo si acceda in modo concorrente da parte di due processi Preleva e Versa e quindi si deve regolare l'accesso in mutua esclusione: completa il codice riportato di seguito in modo da garantire il corretto funzionamento della sincronizzazione sapendo che la banca non permette scoperto sul conto corrente.

```
Risorse condivise
int saldo = 300;
semaphore mutex1=.............;

Processo Preleva() {
   int movimento;
   ..........................
   ..........................
   movimento = saldo;
   movimento = movimento-500;
   saldo     = movimento;
   scrivi ("effettuato prelievo di 500 euro")
..........................
..........................
```

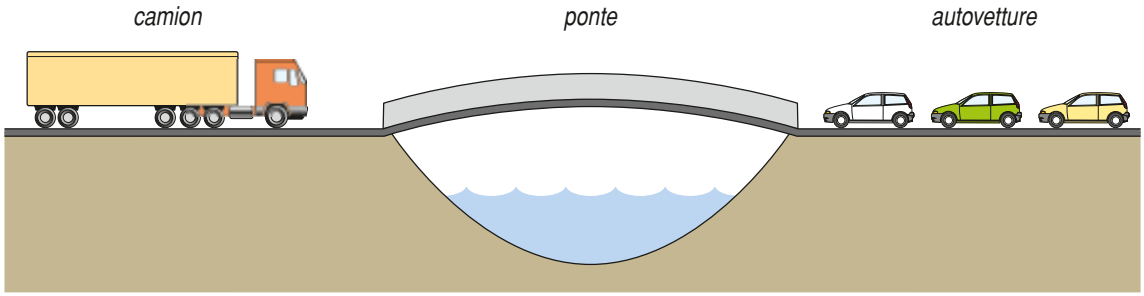
```
}
Processo Versa() {
   int movimento;
   ..............................
   ..............................
   Movimento = saldo;
   movimento = movimento+500;
   conto     = movimento;
   scrivi ("deposito di 500 euro")
   ..............................
   ..............................
```

5 Un cinema con capienza di MAX persone non dispone di un sistema di prenotazione, e quindi un cliente arriva alla biglietteria dove sono presenti due casse, aspetta il suo turno e, se trova un posto libero, può accede alla proiezione.
Al termine della proiezione due hostess intervistano gli spettatori richiedendo un parere espresso con un numero da 1 a 5.
Si richiede di gestire le sincronizzazioni e di scrivere il codice per visualizzare ordinatamente la classifica dei pareri ricevuti.

6 Si deve gestire l'ufficio prenotazioni/cassa di un ospedale: i pazienti prendono un ticket con un numero progressivo da un dispenser in base a 5 categorie di servizio/prestazione, etichettate con le lettere da A a F dove le richieste di tipo A hanno priorità più alta rispetto alle altre, e così via, in ordine decrescente fino alla F.
Sono disponibili tre sportelli che devono smaltire il lavoro in base alla priorità e al numero d'ordine del paziente, garantendo l'assenza di starvation (per esempio, limitando l'attesa massima a 20 utenti).

Scrivi una possibile soluzione in pseudocodifica.

7 Si consideri la situazione rappresentata in figura dove due strade sono unite da un ponte a un unico senso di marcia e quindi il traffico lo attraversa a senso unico alternato.
Non viene stabilita alcuna precedenza, quindi il primo che sopraggiunge al ponte, se lo trova libero oppure se altre macchine lo stanno attraversando nella suo stesso senso di marcia, lo può attraversare: un'autovettura che giunge dalla direzione opposta deve attendere che tutti quelli che sono sul ponte abbiano finito di attraversarlo.
Descrivi al situazione regolando la sincronizzazione in modo tale che al massimo possano passare 100 autovetture consecutivamente per senso di marcia.
Successivamente introduci la condizione che possano sopraggiungere sia autovetture che camion ma che, per il loro peso, questi ultimi debbano attraversare il ponte da soli.



LEZIONE 5

PROBLEMI "CLASSICI" DELLA PROGRAMMAZIONE CONCORRENTE: PRODUTTORI/CONSUMATORI

IN QUESTA LEZIONE IMPAREREMO...

- le caratteristiche del problema produttori/consumatori
- a risolvere il problema produttori/consumatori

Generalità

Esiste un certo numero di problemi "classici" della programmazione concorrente che rappresentano la casistica completa delle diverse situazioni di concorrenza; possono essere suddivisi in tre gruppi, dei quali riportiamo il nome con il quale sono indicati nella letteratura informatica:

- produttore/consumatore (producer/consumer)
  - 1 produttore – 1 consumatore
  - buffer limitato o circolare (bounded buffer)
  - n produttori – n consumatori
- lettori e scrittori (readers/writers)
- filosofi a cena (dining philosophers)

In questa lezione descriveremo il primo dei tre problemi nelle sue diverse situazioni mentre rimandiamo gli altri alle successive lezioni.

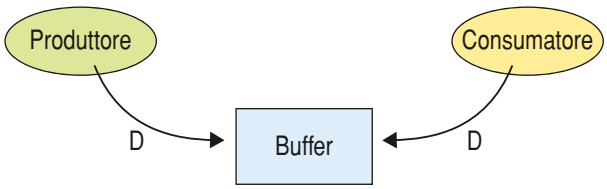
Produttore/consumatore

Il *problema del produttore/consumatore* è un modello classico per lo studio della sincronizzazione nei sistemi operativi dove più processi devono coordinarsi per scambiarsi delle informazioni.

Un produttore e un consumatore

Nella sua formulazione più semplice sono presenti due processi, il primo, detto produttore, che vuole inviare informazioni scrivendo un dato (o un messaggio) in un'area di memoria condivisa (nei

nostri esempi in una variabile) e il secondo, chiamato consumatore, che vuole prelevare i valori prodotti dal primo e "consumarli". ▶



Un tipico esempio è la gestione della stampa, dove vengono inviati i dati al processo che si occupa di stamparli.

Il sistema è soggetto ai seguenti vincoli:
- il *produttore* può depositare un dato alla volta e non può scrivere un nuovo messaggio se il consumatore non ha ancora raccolto il precedente;
- il *consumatore* può leggere un dato alla volta e, naturalmente, solo dopo che sia stato prodotto dal processo produttore e non deve leggere due volte lo stesso valore;
- nel sistema non devono verificarsi situazioni di deadlock.

I due processi devono scambiarsi dei messaggi per comunicarsi rispettivamente la disponibilità del dato da leggere e l'avvenuta lettura di dato e, quindi, la possibilità di produrre il successivo: ciascuno processo deve attendere un segnale da parte dell'altro processo.

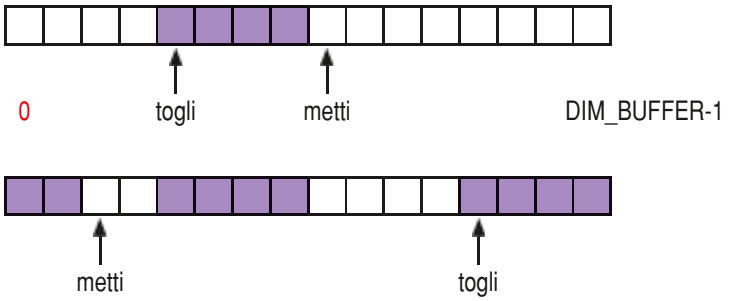
Una possibile soluzione in pseudo codifica che utilizza due semafori è la seguente:

```
semaphore pieno = 0              // all'inizio il buffer non è pieno   (sem.rosso)
semaphore vuoto = 1              // all'inizio il buffer è vuoto       (sem.verde)

int buffer;

Processo produci(int dato)
inizio
   P(vuoto)                      // quando il buffer è vuoto
      buffer = dato
   V(pieno)                      // segnala che il buffer è pieno
fine

Processo consuma()
inizio
   P(pieno)                      // quando il buffer è pieno
      <consuma il dato>
   V(vuoto)                      // segnala che il buffer è vuoto
fine
```

Il funzionamento è abbastanza semplice: vengono utilizzati due semafori, il primo che indica quando il buffer è pieno e quindi verrà settato del produttore, il secondo che indica quando il dato è stato consumato, e sarà quindi settato dal consumatore: alla fine delle operazioni ogni processo deve avvisare l'altro del cambiamento di stato e lo farà resettando il semaforo opportuno:
- semaphore vuoto
  - produttore lo mette a 0 quando inizia a inserire un dato
  - consumatore lo mette a 1 quando ha tolto un dato
- semaphore pieno
  - produttore lo mette a 1 quando ha inserito un dato
  - consumatore lo mette a 0 quando inizia a togliere il dato

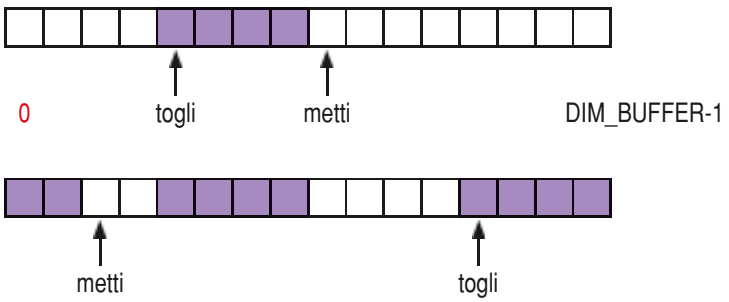
Un produttore e un consumatore e buffer circolare

Nel caso più generale la memoria condivisa può contenere più dati (o messaggi) e viene realizzata per esempio con un buffer circolare implementato con un vettore di lunghezza DIM_BUFFER.

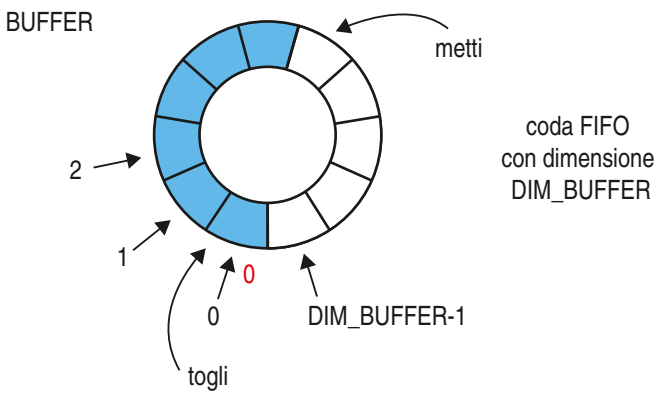


Modifichiamo le condizioni di funzionamento in:

- il produttore non può inserire un messaggio nel buffer se questo è pieno;
- il consumatore non può prelevare un messaggio dal buffer se questo è vuoto.



Introduciamo due puntatori, *metti* e *togli*, che individuano rispettivamente la prima porzione vuota e piena del buffer dove inizialmente viene posto *metti* = *togli*:



I due puntatori *metti* e *togli* che si spostano attorno all'array realizzano una coda circolare FIFO con una dimensione massima prefissata (DIM_BUFFER): vengono utilizzati in modo ciclico (modulo dell'ampiezza del buffer mediante l'operatore %).

Una codifica del processo produttore è la seguente:

```
Processo produci(int dato)
inizio
   P(vuoto)                    // quando il buffer ha almeno uno spazio vuoto
      buffer[metti] = dato;
      // aggiornamento indice: quando si arriva DIM_BUFFER si ritorna a 0
      metti = (metti + 1) % DIM_ BUFFER;
   V(pieno)                    // segnala che il buffer è pieno
```

Una codifica del processo consumatore è la seguente:

```
Processo consuma()
inizio
   P(pieno)                      // quando il buffer è pieno
      <consuma il dato>
      togli =(togli+1)%DIM_BUFFER;
   V(vuoto)                      // segnala che il buffer è vuoto
fine
```

Se utilizziamo i semafori binari questa soluzione però presenta un inconveniente: il produttore scrive il secondo dato solo dopo che il consumatore lo ha prelevato, rendendo di fatto inutile il buffer circolare.
Dato che ogni cella del vettore è una risorsa, siamo in presenza di risorse con molteplicità > 1 ed è quindi necessario modificare il semaforo di accesso al buffer sostituendolo con un semaforo generalizzato che "tenga il conteggio" dei posti liberi nel buffer, così che il produttore possa inserire successivi dati senza che vengano consumati (al limite ne potrà inserire DIM_BUFFER prima che il consumatore inizi a leggere); è inoltre necessario sostituire anche il secondo semaforo sempre con un semaforo generalizzato in modo da "ricordare" quanti nuovi dati non ancora letti sono presenti nel buffer.
È sufficiente introdurre la modifica dell'inizializzazione del programma:

```
semaphore vuoto = DIM_BUFFER     // all'inizio il buffer ha posti liberi
semaphore pieno = 0              // dati da leggere presenti nel buffer
buffer array[DIM_BUFFER]
```

Più produttori e più consumatori

Estendiamo ora al caso generale, dove possono essere presenti più produttori e più consumatori: questa nuova situazione può comportare nuove necessità di sincronizzazione in quanto ora non è sufficiente la sincronizzazione appena descritta, poiché abbiamo due tipi di problematiche:

- sincronizzare i produttori e i consumatori, così come nel caso precedente, verificando che se qualunque dei produttori volesse produrre ma non ci fosse la disponibilità di un buffer libero, questo deve sospendersi e attendere che uno qualunque dei consumatori consumi un dato liberando così lo spazio di un buffer;
- sincronizzare gli n produttori fra di loro e gli m consumatori fra di loro perché in questa condizione anche le variabili togli e metti sono condivise e quindi divengono di fatto risorse comuni.

Se per esempio due produttori contemporaneamente effettuano la P(V) ed entrano nella regione critica, molto probabilmente scriveranno nella stessa cella ed eseguiranno in modo errato l'operazione:

```
metti = (metti + 1) % DIM_BUFFER;
```

Dobbiamo risolvere questo conflitto nei produttori così come lo ritroviamo simile anche nei lettori, che potrebbero leggere contemporaneamente lo stesso dato.

Inizialmente definiamo i semafori e il buffer comune:

```
semaphore vuoto = DIM_BUFFER     // all'inizio il buffer ha posti liberi
```

```
semaphore pieno = 0              // dati da leggere presenti nel buffer
buffer array[DIM_BUFFER]
```

Per i produttori useremo quindi una procedura come quella di seguito riportata, che dopo aver accertata l'esistenza di almeno un buffer libero, serializza tra i produttori l'accesso alla variabile metti in modo da garantire la mutua esclusione al suo accesso da parte dei produttori che richiedono di aggiornarla.

Questa operazione equivale a inserire una regione critica.

```
Processo produci(int dato)
semaphore mutexP = 1              // semaforo che regola i produttori
int tempo;
inizio
 P(vuoto)                         // quando il buffer ha spazio libero
  P(mutexP)                       // blocca l'accesso alla RC agli altri produttori
   tempo = metti                  // salva il valore corrente di metti
   metti =(metti+1)% DIM_BUFFER;  // aggiornamento dell'indice
  V(mutexP)                       // libera l'accesso alla RC agli altri produttori
  buffer[metti] = dato            // inserisci il nuovo dato
 V(pieno)                         // segnala che il buffer è pieno
```

Per i consumatori useremo quindi una procedura simile, di seguito riportata:

```
Processo consuma()
int tempo;
semaphore mutexC = 1              // semaforo che regola i consumatori
inizio
 P(pieno)                         // quando il buffer è pieno
  P(mutexP)// blocca l'accesso alla RC agli altri produttori
   tempo =togli                   // salva il valore corrente di metti
   togli =(togli+1)%DIM_BUFFER;
  V(mutexP)// libera l'accesso alla RC agli altri produttori
  <consuma il dato>
 V(vuoto)                         // segnala che il buffer è vuoto
fine
```

Osserviamo come l'operazione di scrittura messaggio nel buffer non sia stata serializzata in quanto più produttori contemporaneamente devono poter fare il proprio "deposito": per serializzare anche questa operazione basta scambiare tra loro i semafori, cioè testare prima il mutexP/mutexC e successivamente i semafori pieno/vuoto.

Bisogna stare attenti a non precludere il parallelismo del riempimento del buffer da parte dei produttori, soprattutto se le operazioni di scrittura nel buffer sono piuttosto lente: se così fosse, tutti gli altri produttori rimarrebbero in attesa per lungo tempo anche in presenza di posti liberi, e anche tutti i consumatori rimarrebbero fermi, non avendo nulla da "consumare".

Per questo motivo sono state portate al di fuori della regione critica le operazioni di deposito e di prelievo dei dati.

Verifichiamo le competenze

1 Scrivi una funzione che restituisca il numero di celle correntemente occupate nel buffer, nell'ipotesi che non sia disponibile una primitiva per interrogare il valore dei semafori a conteggio. (Suggerimento: non effettuare incrementi circolari dei due indici togli e metti e gli accessi circolari alle celle del buffer).

2 In un sistema i tempi di produzione e consumazione sono casuali ma della medesima durata massima per produttori e consumatori.
Descrivi come cambia il tempo di attesa passiva sulle quattro invocazioni della primitiva P() al variare del
- del numero di produttori;
- del numero di consumatori;
- delle dimensioni del buffer.

3 Si vuole realizzare un sistema produttori/consumatori dove le operazioni di inserimento e prelievo risultino non-bloccanti, ovvero dove l'operazione di inserimento su un buffer pieno e di prelievo da un buffer vuoto invece che causare la sospensione restituisca immediatamente un errore. Impostare una soluzione a questo problema utilizzando i semafori di Dijkstra.
Si consideri il seguente pseudocodice:

```
mutex = 1
pieno = 0
vuoto =10
Processo P1(){
   while(true){
      elemento=produci();
      P(mutex);
      P(pieno);
      inserisci(elemento);
      V(vuoto);
      V(mutex);
   }
}

Processo P2(){
   while(true){
      P(mutex);
      P(vuoto);
      valore=rimuovi();
      V(pieno);
      V(mutex);
      consuma(valore);
   }
}
```

I due processi concorrenti si passano i dati attraverso un buffer condiviso con il meccanismo del produttore/consumatore. Il semaforo 'mutex' è inizializzato a 1, il semaforo 'pieno' è inizializzato a 0 e 'vuoto' è inizializzato a 10, che è la dimensione del buffer.
Discuti il funzionamento della sincronizzazione e, nel caso ci sia un errore di programmazione o di inizializzazione, proponi una soluzione corretta.

LEZIONE 6

PROBLEMI "CLASSICI" DELLA PROGRAMMAZIONE CONCORRENTE: LETTORI/SCRITTORI

IN QUESTA LEZIONE IMPAREREMO...

- le caratteristiche del problema lettori/scrittori
- a risolvere il problema lettori/scrittori

Problema dei lettori e degli scrittori

In un sistema sono presenti due gruppi di processi, i lettori e gli scrittori:

- i lettori non possono modificare i dati, ma solo prelevare in modo non distruttivo il contenuto di un buffer per elaborarlo;
- gli scrittori possono invece produrre un dato e quindi aggiornare il contenuto del buffer.

La situazione è diversa da quella analizzata nella lezione precedente dei produttori/consumatori in quanto in questo caso più lettori possono contemporaneamente utilizzare un dato senza "danneggiarlo": quindi un lettore può accedere a un dato anche se è già presente un altro lettore, mentre lo scrittore deve accedere in modo esclusivo come nell'esempio della lezione precedente.
Esiste un vincolo di mutua esclusione fra lettori e scrittori nell'uso della risorsa e inoltre gli scrittori tra loro si contendono la risorsa, che quindi deve essere gestita in mutua esclusione.

Esistono diverse soluzioni a questo problema, in base alle ipotesi che vengono fatte sulle possibili situazioni di funzionamento.

Prima soluzione

Ipotizziamo che:

- i lettori possano accedere alla risorsa anche se ci sono scrittori in attesa;
- uno scrittore possa accedere alla risorsa solo se questa è libera.

Anche se l'ipotesi fatta sembra sensata, analizzandola attentamente ci accorgiamo che potrebbe portare a una situazione di starvation da parte degli scrittori, in quanto se i lettori continuano ininterrottamente a leggere, non potranno mai accedere in scrittura sul buffer.

Questa soluzione viene lo stesso implementata considerando il fatto che la starvation si verifica raramente anche considerando le velocità relative dei lettori rispetto agli scrittori, la frequenza delle operazioni e il tempo necessario a eseguirle.

Le procedure necessarie a realizzare la sincronizzazione di questo sistema sono quattro:
- una di richiesta lettura;
- una di fine lettura;
  – fra queste due si interporrà l'operazione di lettura vera e propria;
- una di richiesta scrittura;
- una di fine scrittura;
  – fra queste si interporrà l'operazione di scrittura.

Introduciamo una variabile `numLettori` necessaria per conoscere il numero di lettori che contemporaneamente stanno utilizzando il buffer comune, che sono cioè all'interno della Regione Critica lettura/scrittura: questa variabile deve essere modificata in competizione fra i lettori e per un suo corretto funzionamento deve essere manipolata (incrementata e decrementata) in mutua esclusione regolata dal semaforo mutex.

```
semaphore mutex = 1          // regola l'accesso alla variabile numLettori
semaphore sincro = 1         // regola l'accesso al buffer per scrivere/leggere
numLettori = 0               // numero di lettori all'interno della RC
```

```
procedura inizioLettura
inizio
   P(mutex)                  // mutua esclusione accesso al contatore
      numLettori++           // aggiornamento numero lettori entrati nella RC
      if (numLettori=1)      // se è il primo lettore che accede al buffer
         then P(sincro)      // blocca gli scrittori
   V(mutex)                  // libera l'accesso alla RC agli altri produttori
fine
```

Analizziamo il funzionamento dei due semafori, mutex e sincro:
- mutex serve per regolare i lettori che possono accedere contemporaneamente ai dati; per poter incrementare il contatore deve accedere in modo esclusivo e bloccare gli altri lettori;
- sincro serve per sincronizzare i lettori dagli scrittori: il primo lettore che riesce ad accedere al buffer deve verificare o meno la presenza di un scrittore nella RC di lettura/scrittura e quando riesce ad accedere blocca gli solo gli scrittori mettendo a rosso il semaforo sincro; non blocca i successivi lettori perché questa istruzione viene eseguita solo quando si verifica `numLettori=1`.

```
procedura fineLettura
inizio
   P(mutex)                  // mutua esclusione accesso al contatore
      numLettori--           // aggiornamento numero lettori entrati nella RC
      if (numLettori)=0      // nessun lettore dentro la RC
         then V(sincro)      // sblocca gli scrittori
   V(mutex)                  // libera l'accesso al contatore ai lettori
fine
```

La funzione di rilascio è molto simile: mutex regola sempre l'accesso al contatore e quando l'ultimo lettore sta lasciando la RC di lettura/scrittura metterà a verde il semaforo sincro in modo da "liberare" l'accesso a chiunque voglia successivamente accedere ai dati.
Quindi sincro non viene sbloccato ogni volta che un lettore esaurisce il suo lavoro, ma solo quando *tutti* i lettori hanno terminato.

```
procedura inizioScrittura
inizio
   P(sincro)                  // blocca sia scrittori che lettori
fine
```

```
procedura fineScrittura
inizio
   V(sincro)                   // sblocca sia scrittori che lettori
fine
```

Le procedure di inizioScrittura e fineScrittura si realizzano con una sola istruzione sul semaforo sincro, che regola l'accesso al buffer di lettura/scrittura; quando un processo inizia a scrivere lo mette a rosso bloccando ogni altro accesso e quando finisce di scrivere lo mette a verde permettendo a qualsiasi altro processo di accedervi.

Queste funzioni devono essere chiamate all'interno dei processi che, nel caso di buffer circolare, devono inoltre provvedere anche alla sincronizzazione della risorsa finita, come descritto nell'esempio dei produttori/consumatori.

Poniamoci ora due domande:

1. quando un lettore rimane in attesa?
   Un lettore si sospende per due motivi:
   - A perché c'è un processo scrittore che sta scrivendo;
   - B perché un altro lettore ha chiamato inizioLettura e quindi ha chiamato in causa la regione critica relativa alla variabile numLettori.
2. che cosa succede se entra prima un lettore?
   Dipende dalla applicazione: se i lettori devono leggere qualcosa che hanno scritto gli stessi processi scrittore, allora non funziona, se invece stiamo, per esempio, regolando l'accesso a un archivio già esistente, non ci sono problemi.

La soluzione è comunque sbilanciata a favore dei lettori dato che, potendo accedere contemporaneamente, potrebbero provocare una lunga lista di attesa degli scrittori che devono attendere che non sia presente più nessun lettore.

Seconda soluzione

Una seconda soluzione che migliora "la vita" agli scrittori e lettori è quella presentata di seguito dove uno scrittore, prima di terminare le sue operazioni, cerca di "passare il testimone" a un altro scrittore invece che ai lettori.

```
semaphore mutex   = 1         // regola l'accesso alla variabile numLettori
semaphore mutex2  = 1         // regola l'accesso alla variabile numScrittori
```

```
semaphore mutex1  = 1        // regola l'accesso degli scrittori uno alla volta
semaphore sincro  = 1        // regola l'accesso al buffer per scrivere/leggere
numLettori        = 0        // numero di lettori all'interno della RC
numScrittori      = 0        // numero di scrittori all'interno della RC
```

La procedura inizioScrittura è praticamente identica alla precedente procedura inizioLettura in modo da contare il numero degli scrittori che desiderano accedere alla RC.

```
procedura inizioScrittura
inizio
   P(mutex2)                 // mutua esclusione accesso al contatore
   numScrittori++            // aggiornamento numero scrittori entrati nella RC
   if (numScrittori=1)       // se è il primo lettore che accede al buffer
      then P(sincro)         // blocca i lettori
   V(mutex2)                 // libera l'accesso al contatore altri scrittori
   P(mutex1)                 // blocca l'accesso alla RC agli altri scrittori
fine
```

Il semaforo mutex2 serve per garantire la mutua esclusione sulla variabile numScrittori così come il semaforo mutex regolava l'accesso alla variabile numLettori nel caso prima descritto.

Il semaforo sincro ha lo stesso compito della prima soluzione, cioè realizza la mutua esclusione fra la 'categoria' lettori e la 'categoria' scrittori: l'unica differenza rispetto alla prima soluzione è che mentre i lettori possono essere contemporaneamente dentro la RC lettura/scrittura, gli scrittori devono susseguirsi uno alla volta, cioè in mutua esclusione.
Questo accesso serializzato viene realizzato con un ulteriore semaforo, mutex1, che viene posto a rosso quando il primo scrittore sta scrivendo e successivamente messo a verde dalla procedura di fineScrittura.

```
procedura fineScrittura
inizio
   V(mutex1)                 // libera l'accesso agli eventuali scrittori
   P(mutex2)                 // mutua esclusione accesso al contatore
      numScrittori--         // aggiornamento numero lettori entrati nella RC
      if(numScrittori=0)     // nessuno scrittore ne dentro la RC ne in attesa
         then V(sincro)      // sblocca i lettori
   V(mutex2)                 // libera l'accesso al contatore agli scrittori
fine
```

Ora il problema è diventato doppio: possono bloccarsi sia i lettori che gli scrittori, quindi non abbiamo ancora risolto il problema.
La soluzione ottimale è quella di alternare lettori e scrittori in modo che entrambe le "categorie" possano evolvere.

Verifichiamo le competenze

1 Si consideri una variante del problema dei lettori scrittori con i seguenti due lettori:

```
Processo Lettore1(){
   while(true){
      P(mutex);
      nrlettori++;
      if(nrlettori==1)
         printf("primolettore");
      V(mutex);
      leggi_archivio();
      P(mutex);
      nrlettori--;
      if(nrlettori==0)
         printf("ultimolettore");
      V(mutex);
   }
}

Processo lettore2(){
   while(true){
      P(mutex);
      nrlettori++;
      if(nrlettori==1)
         printf("primolettore");
      V(mutex);
      leggi_archivio();
      P(mutex);
      nrlettori--;
      if(nrlettori==0)
         printf("ultimolettore");
      V(mutex);
   }
}
```

Descrivi il funzionamento indicando se il meccanismo è "equo" oppure privilegia una categoria di processi.

Volendo riscriverla su di una piattaforma che non ha le primitive semaforiche, come si possono realizzare le sezioni critiche? Modifica il codice in tale senso.

2 Situazione: la Toilette Unisex

Un ristorante ha un'unica toilette sia per uomini che per donne: realizza un'applicazione concorrente nella quale ogni utente della toilette (uomo o donna) è rappresentato da un processo e il bagno come una risorsa.
La politica di sincronizzazione tra i processi dovrà garantire che:

- nella toilette non vi siano contemporaneamente uomini e donne;
- nell'accesso alla toilette, le donne abbiano la priorità sugli uomini.

Si supponga che la toilette abbia una capacità limitata a N persone.

3 Situazione: ponte con utenti grassi e magri

A un ponte pedonale che collega le due rive di un fiume possono accedere due tipi di utenti: utenti magri e utenti grassi.
Il ponte ha una capacità massima MAX che esprime il numero massimo di persone che possono transitare contemporaneamente su di esso ma è talmente stretto che il transito di un grasso in una particolare direzione impedisce l'accesso al ponte di altri utenti (grassi e magri) in direzione opposta.
Realizza una politica di sincronizzazione delle entrate e delle uscite dal ponte che tenga conto delle specifiche date e che favorisca gli utenti magri rispetto a quelli grassi nell'accesso al ponte.

4 Situazione: BAR dello stadio

In uno stadio è presente un unico bar a disposizione di tutti gli spettatori che assistono alle partite di calcio. Oltre ai giornalisti, al bar accedono sia i tifosi della squadra locale che quelli della squadra ospite.
Il bar ha una capacità massima NRMAX di posti, che esprime il numero massimo di persone che il bar può accogliere contemporaneamente: per motivi di sicurezza, nel bar non è consentita la presenza contemporanea di tifosi di squadre opposte.
Il bar chiude per le operazioni di pulizia dopo dieci minuti dall'inizio della partita, riapre prima dell'intervallo, quindi chiude dopo dieci minuti dall'inizio del secondo tempo e riapre prima della fine della partita: nei periodi di chiusura potrebbero essere presenti alcune persone nel bar e in questo caso il barista attenderà l'uscita delle persone presenti nel bar, prima di procedere alla pulizia.
Scrivi le classi opportune necessarie per coordinare barista, giornalisti e tifosi delle opposte fazioni facendo in modo che si dia la precedenza di accesso al bar ai tifosi della squadra ospite.

5 Situazione: barbiere dormiente

In un salone lavora un solo barbiere, ci sono NRSEDIE=5 sedie per accogliere i clienti in attesa e una sedia di lavoro per il servizio dei clienti.
Il barbiere e i clienti sono processi e la poltrona è una risorsa, che può essere assegnata a un cliente per il taglio dei capelli, oppure utilizzata dal barbiere per dormire: infatti, se non ci sono clienti, il barbiere si addormenta sulla sedia di lavoro.
Un cliente quando arriva deve svegliare il barbiere, se addormentato, o accomodarsi su una delle sedie in attesa che finisca il taglio corrente. Se nessuna sedia è disponibile, il cliente preferisce non aspettare e lascia il negozio.
Dopo che un cliente ha occupato la poltrona, il barbiere esegue il taglio dei capelli e al termine:

- se ci sono clienti in attesa del proprio turno, riattiva il primo;
- altrimenti occupa la poltrona e si addormenta.

Scrivi le classi opportune necessarie per coordinare il barbiere e i clienti.

LEZIONE 7

PROBLEMI CLASSICI DELLA PROGRAMMAZIONE CONCORRENTE: DEADLOCK, BANCHIERE E FILOSOFI A CENA

IN QUESTA LEZIONE IMPAREREMO...

- il concetto di deadlock
- a riconoscere le situazioni di deadlock
- a risolvere le situazioni di deadlock
- l'algoritmo del banchiere proposto da Dijkstra

Perché si genera un deadlock

Con deadlock (o stallo) indichiamo quelle situazioni nella quali due (o più) processi si ostacolano a vicenda impedendosi reciprocamente di portare a termine il proprio lavoro: il deadlock viene anche chiamato "abbraccio mortale" in quando l'interferenza porta al fallimento di entrambi e… alla loro morte.

> Oltre a loro stessi, i processi "coinvolti" nel deadlock ostacolano anche gli eventuali altri processi presenti nel sistema, in quanto le risorse a loro allocate non possono essere utilizzate da nessuno.

Non tutti i problemi di concorrenza possono portare a situazioni di deadlock: affinché ci sia un deadlock è necessaria la presenza di:

1. mutua esclusione: le risorse coinvolte devono essere seriali, cioè ogni risorsa o è assegnata a un solo processo o è libera;
2. assenza di prerilascio (preemption): le risorse coinvolte non possono essere prerilasciate, ovvero devono essere rilasciate volontariamente dai processi che le controllano prima cha abbiano terminato di usarle;
3. richieste bloccanti (detta anche "◀ hold and wait ▶"): le richieste devono essere bloccanti, e un processo che ha già ottenuto risorse può chiederne ancora;
4. attesa circolare: devono essere presenti nel sistema almeno due processi e ciascuno di essi è in attesa di una risorsa occupata dall'altro.

> Affinché si verifichi un deadlock nel sistema queste condizioni devono verificarsi contemporaneamente: quindi sono condizioni necessarie e sufficienti (dimostrato da Coffman nel 1971).

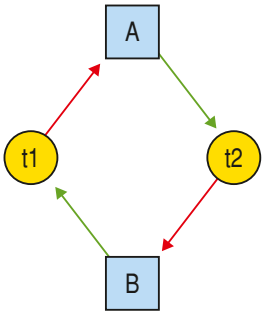
In questa lezione analizzeremo le tecniche che possono essere utilizzate per gestire il problema dei deadlock.

◀ **Hold and wait** A process or thread holding a resource while waiting to get hold of another resource. ▶

Individuazione dello stallo

Per individuare le situazioni di stallo possiamo utilizzare i grafi di Holt, o grafi di allocazione (Resource Allocation Graphs RAG), descritti nella lezione 2 dell'unità di apprendimento 1: con questi è possibile verificare se una data sequenza di richieste-acquisizioni porta allo stallo.

Analizziamo la situazione riportata nel grafo della figura a lato: ▶

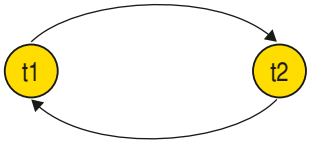


Lo interpretiamo nel modo seguente:
- i due processi (o thread) t1 e t2 sono in attesa di bloccare rispettivamente la risorsa A e B;
- le risorse A e B sono allocate rispettivamente ai thread t2 e t1.

Osserviamo come:

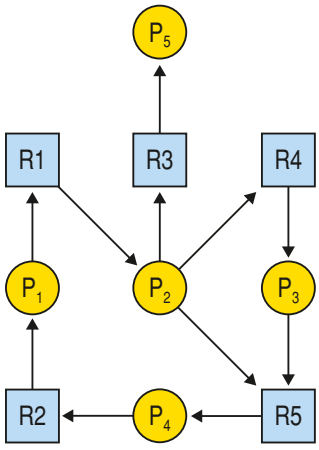
> t1 richiede una risorsa che è occupata da t2 e t2 richiede una risorsa che è occupata da t1!

Una variante del grafo di Holt, detto grafo wait-for (o di attesa), si ottiene eliminando i nodi di tipo risorsa e "unendo" gli archi appropriati entranti-uscenti dalla risorsa eliminata: la situazione di blocco precedente risulta quindi descritta dal grafo di attesa a lato: ▶

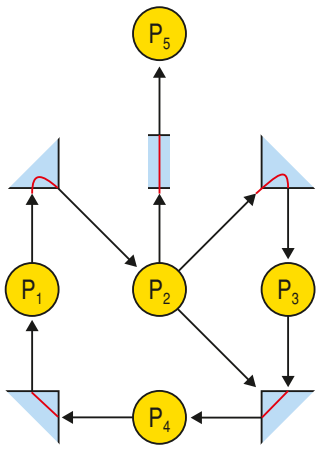


ESEMPIO 12 *Passaggio da grafo di allocazione a grafo di attesa*

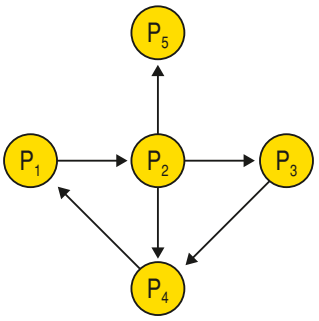
Trasformiamo un grafo RAG in un grafo di attesa:



Eliminiamo le risorse "collegando" tra loro le frecce che escono da un processo e, "attraversando" una risorsa, entrano in un altro processo:



Il grafo risultante è il seguente:



La situazione di deadlock è rappresentata dall'esistenza di un **ciclo** in un grafo di attesa (è presente una situazione di attesa circolare).

Il grafo di Holt contiene un ciclo se e solo se il grafo wait-for contiene un ciclo.

Il grafo di Holt ci permette di individuare o escludere i deadlock grazie ai seguenti teoremi:

I TEOREMA SUL GRAFO DI HOLT

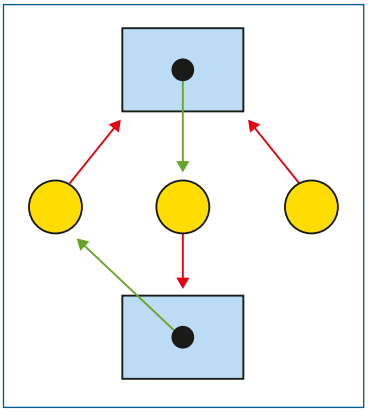
Se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili lo stato è di deadlock se e solo se il grafo di Holt contiene un ciclo.

II TEOREMA SUL GRAFO DI HOLT

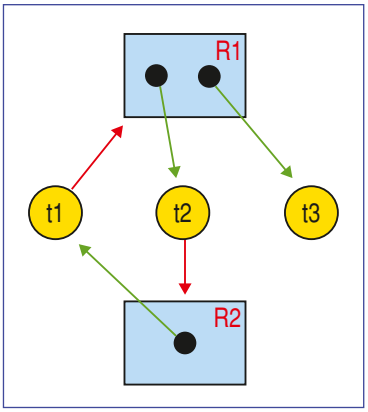
Se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili lo stato non è di deadlock se e solo se il grafo di Holt è completamente riducibile, cioè esiste una sequenza di passi di riduzione che elimina tutti gli archi del grafo.

Nel caso in cui le risorse sono presenti con molteplicità superiore a 1, la presenza di un ciclo nel caso di Holt non è condizione sufficiente per avere deadlock.

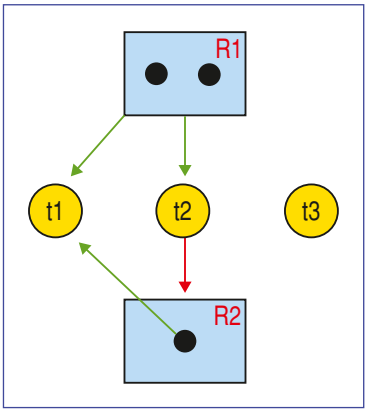
Lo possiamo verificare con i seguente esempi:



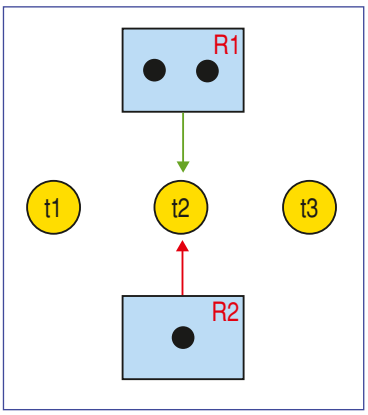
In questa situazione la presenza di un ciclo indica la *presenza di un deadlock*. ▶



In questa situazione anche se è presente un ciclo *non sussiste la situazione di deadlock* in quanto possiamo ridurre il grafo, dato che il thread t3 può evolvere e quindi prima o poi rilascerà la risorsa R1 che verrà assegnata al thread t1: ▶



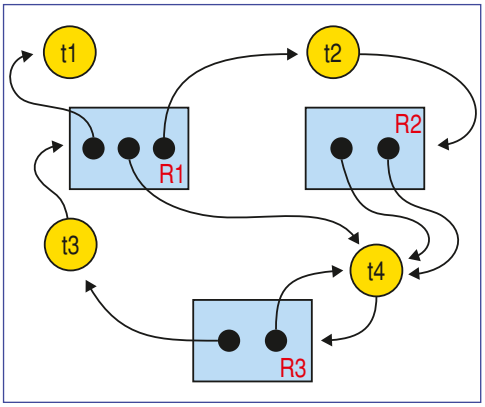
Questo potrà evolvere e rilasciare entrambe le risorse al thread t2 che può portare a termine le sue elaborazioni. ▶



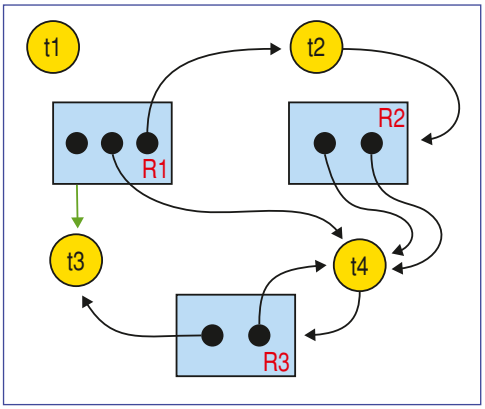
ESEMPIO 13

Riduciamo un grafo-complesso

Analizziamo un esempio più articolato per scoprire se è possibile che si verifichi un deadlock. ▶

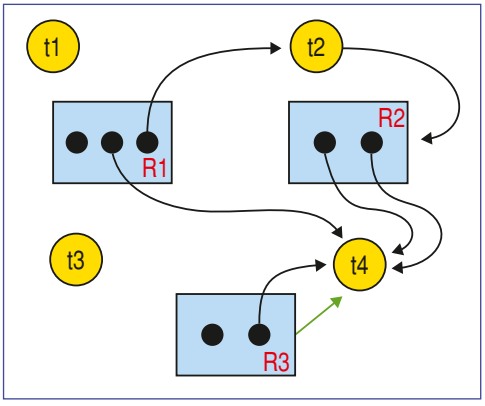


Per prima cosa osserviamo che il thread t1 ha assegnato l'unica risorsa che richiede e quindi è in grado di portare a termine il proprio lavoro, quindi lo riduciamo: ▶



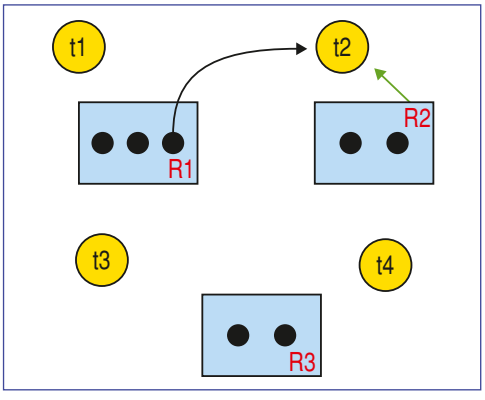
Dopo aver effettuato la riduzione di t1, possiamo riassegnare la R1 da lui posseduta a t3.

Ora è il thread t3 che può evolvere e quindi lo riduciamo: ▶



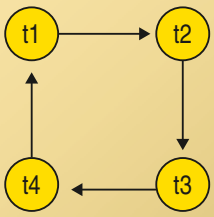
Dopo aver effettuato la riduzione di t3, possiamo riassegnare la R3 da lui posseduta a t4.

A questo punto anche il t4 ha a disposizione tutte le risorse che gli necessitano, e lo possiamo ridurre: ▶



Dopo la riduzione di t4 possiamo effettuare anche la riduzione di t2: quindi il grafo è stato completamente ridotto, come richiesto dal II teorema, e quindi possiamo affermare con certezza l'assenza dello stallo.

Le situazioni di deadlock possono coinvolgere anche più di 2 thread, come mostrato dal grafo di attesa riportato di seguito, dove esiste un ciclo che coinvolge 4 thread.



Come affrontare lo stallo

Lo stallo può essere affrontato sostanzialmente in quattro diverse strategie:
- detection e recovery: riconoscerlo ed eliminarlo;
- avoidance: evitarlo con specifiche politiche di allocazione;
- prevention: impedirlo rimuovendo una delle quattro condizioni necessarie affinché si verifichi;
- ignorare il problema.

Eliminare lo stallo: detection e recovery

Il sistema deve essere continuamente monitorato per riconoscere le situazioni stallo, utilizzando i grafi di Holt, e quando si individua un problema si interviene per eliminarlo.

La risoluzione delle situazioni indesiderate può avvenire in modo:
- manuale: è richiesto all'operatore di intervenire per risolvere la situazione a favore di uno dei processi in modo da sbloccare il sistema;
- automatica: il sistema operativo è in grado di risolvere automaticamente lo stallo applicando meccanismi opportuni che utilizzano specifiche politiche prestabilite.

Le soluzioni per entrambe le modalità sono le seguenti:
- terminazione dei processi: è il modo più semplice e allo stesso tempo più drastico per risolvere la situazione di stallo: si forza la terminazione di un processo alla volta (*terminazione parziale*) osservando se man mano viene risolta la situazione di stallo e, in caso negativo, si procede alla terminazione di tutti i processi (*terminazione totale*) e si sceglie quale deve essere il primo processo che deve essere fatto ripartire;
- prerilascio di una risorsa (preemption): viene forzato il prerilascio di una risorsa da parte di uno dei processi in stallo in modo che la possa acquisire un processo che possa evolvere grazie a quella allocazione; questa operazione, però, non può essere effettuata per tutti i tipi di risorse (si pensi per esempio a una stampante);
- checkpoint/rollback: viene fatto periodicamente il salvataggio dello stato dei processi in determinati istanti ben definiti (checkpoint) e viene salvato su disco: in caso di deadlock, si ripristina (rollback) uno o più processi a uno stato precedente, fino a quando il deadlock non scompare.

Due osservazioni sui metodi sopra descritti:
- *terminare i processi* può essere costoso: per esempio se "uccidiamo" un processo che è in esecuzione da parecchio tempo verrebbe "ignorato e perso" completamente tutto il lavoro da esso effettuato; inoltre, se il processo viene terminato in una sezione critica, si rischia di lasciare le risorse in uno stato inconsistente;
- *fare preemption* non sempre è possibile in automatico e può richiedere interventi manuali.

Evitare lo stallo: avoidance

Si cerca di evitare lo stallo analizzando in anticipo l'utilizzo che il processo farà delle risorse che richiederà nella sua evoluzione in modo da controllare se tra queste operazioni può sorgere il pericolo del verificarsi di un deadlock: se questo può avvenire, si ritarda l'esecuzione di questo processo.

Per individuare se un processo può portare al deadlock si introduce il concetto di stato sicuro e sequenza sicura di esecuzione dei processi:

STATO SICURO

Un sistema è in uno stato sicuro (save) soltanto se esiste una sequenza di completamento sicura per i propri processi.

In questo caso si possono allocare le risorse per ogni processo della sequenza in un ordine preciso e continuare a evitare un **deadlock**.

SEQUENZA SICURA

Una sequenza di processi *<P1, P2, …, Pn>* è sicura se per ogni Pi le richieste di risorse che Pi può fare possono essere soddisfatte da:

- risorse attualmente disponibili;
- risorse detenute da tutti i processi che lo precedono nella sequenza.

Infatti se un processo richiede solo risorse che vengono utilizzate da processi che lo precedono nella sequenza, quando sarà il suo turno le risorse saranno disponibili per la sua esecuzione.

Se il sistema è in uno **stato sicuro** non ci sarà mai la possibilità di deadlock, mentre se non lo è questo potrebbe verificarsi: quindi per evitare il deadlock occorre assicurarsi che il sistema non entri mai in uno stato non sicuro.

Esistono algoritmi che garantiscono la permanenza sempre in stati sicuri e tra tutti ricordiamo il cosiddetto algoritmo del banchiere, proposto da Dijkstra nel 1965.
Il nome deriva dal metodo utilizzato da un ipotetico banchiere che ha un capitale fisso e deve gestire un gruppo prefissato di clienti che richiedono del credito: non tutti i clienti avranno bisogno dello stesso credito simultaneamente ma tutti devono dichiarare in anticipo al banchiere la massima somma della quale hanno necessità che, necessariamente, deve essere inferiore al capitale posseduto dal banchiere e può essere richiesta in una sola volta o in più volte.

La traduzione informatica del problema è che un insieme di processi richiede al gestore il massimo numero di risorse in anticipo.

Le operazioni che possono eseguire i clienti sono due:

- **A** chiedere un prestito una o più volte, ma con somma totale massima non superiore a quanto dichiarato in anticipo;
- **B** restituire il prestito in un tempo finito.

Come si regola il banchiere per dare i prestiti in modo da riuscire a soddisfare tutti i clienti e facendo loro aspettare la disponibilità del denaro in ogni caso in un tempo finito?

Quando un cliente richiede un prestito, prima di concederlo il banchiere controlla se questa operazione può portare la banca in uno stato sicuro e solo in questo caso la richiesta viene accettata.
Per la banca si considera uno stato sicuro la situazione per cui i soldi rimanenti in cassa, dopo aver soddisfatto la richiesta corrente, permettono di soddisfare almeno una successiva richiesta massima da parte di un cliente che ancora non ne ha fatte: in questo modo si garantisce che sempre almeno un altro cliente possa essere servito completamente e quindi in un tempo finito restituirà i soldi in modo da poterli prestare agli altri clienti.
Si può facilmente verificare che se si soddisfa questa condizione l'insieme degli stati generano una sequenza sicura, e quindi è garantito che non ci saranno situazioni di deadlock.

> In questo algoritmo avviene la richiesta di una risorsa singola: se invece si ipotizza che la banca possa fare prestiti in diverse valute si ottiene il caso generale di sistema con classi di risorse multiple.

Nei sistemi operativi questo algoritmo viene così applicato:
ogni processo deve dichiarare il massimo numero di risorse che gli sono necessarie e a ogni richiesta di una nuova risorsa l'algoritmo deve verificare cosa succede nel caso in cui venga soddisfatta questa richiesta, cioè se questa porta il sistema a uno stato sicuro o insicuro: quindi si calcola la quantità di risorse rimanenti nel caso che questo processo venga servito e si valuta se queste risorse possono soddisfare la massima richiesta di almeno un processo che ancora deve essere servito: in tal caso l'allocazione viene accordata, altrimenti viene negata.

Prevenire lo stallo: prevention

Con "prevenzione dello stallo" intendiamo le operazioni che ci permettono di evitare che si verifichi il deadlock intervenendo sulle quattro condizioni necessarie che lo provocano eliminandone una.

Le metodologie utilizzate per la prevenzione sono:

Eliminare la condizione di "risorse seriali"

Questa tecnica si realizza tramite strumenti che permettono di escludere la necessità di **mutua esclusione** permettendo la condivisione di risorse. Un esempio classico è quello che viene realizzato sulle stampanti mediante gli spool, dove ogni processo "crede" di avere la stampante ma in realtà si trova sempre in una situazione di attesa: il problema, di fatto, non viene eliminato ma "spostato" su di un'altra risorsa.
Inoltre questa tecnica non è di possibile realizzazione per tutte le tipologie di risorse.

Eliminare la condizione di "hold & wait"

Per escludere che un processo si impossessi di una risorsa e la mantenga occupata quando è in attesa di una seconda risorsa si provvede a effettuare quella che si chiama allocazione totale**,** cioè si impone che un processo richieda tutte le risorse all'inizio della computazione.

Anche questo meccanismo in generale non è sempre realizzabile in quanto spesso i processi non sanno dall'inizio di quali risorse necessitano e, inoltre, riservandole tutte per un processo si provoca l'arresto di altri processi che magari necessitano solo di una risorsa e potrebbero portare "indisturbati" a compimento il proprio lavoro: si riduce quindi il parallelismo introducendo problemi di possibile starvation.

Effettuare la preemption

Si obbliga un processo a rilasciare le risorse che possiede quando ne richiede un'altra che in quel momento non è disponibile e il processo sarà fatto ripartire soltanto quando può riguadagnare sia le risorse precedentemente possedute sia quelle che sta richiedendo.

Anche questa soluzione spesso non è realizzabile, come per esempio nel caso che la prima risorsa posseduta da un processo sia la stampante e il processo sia già a metà stampa: cosa potrebbe succedere se fosse forzato a cederla?

Una possibile alternativa è quella di controllare quando un processo richiede una risorsa occupata se il processo che la sta occupando in quel momento stia evolvendo oppure attendendo a sua volta una ulteriore risorsa: solo in questo caso si forza il rilascio così da sbloccare il nuovo processo.

Con la prevention il deadlock viene eliminato strutturalmente.

Eliminare la condizione di attesa circolare

Si introduce il concetto di allocazione gerarchica delle risorse attribuendo alle classi di risorse dei valori di priorità e imponendo che ogni processo in ogni istante possa allocare solamente risorse di priorità superiore a quelle che già possiede: se invece ha bisogno di una risorsa a priorità inferiore, deve prima rilasciare tutte le risorse con priorità uguale o superiore a quella desiderata.
In questo modo le risorse devono essere richieste seguendo un ordine prestabilito che viene opportunamente definito dall'amministratore del sistema facendo in modo che siano impossibili i deadlock.
Si può semplicemente verificare che questo meccanismo è efficace in quanto previene il deadlock ma introduce gravi rallentamenti portando il sistema a una alta inefficienza: l'indisponibilità di una risorsa ad alta priorità ritarda processi che già detengono risorse ad alta priorità.

Ignorare il problema

L'ultima soluzione per risolvere il problema dello stallo è quella di applicare l'algoritmo detto "dello struzzo", cioè di "nascondere la testa sotto la sabbia" e di fare finta che non esista il problema, cioè ipotizzare che i deadlock non si possano mai verificare.

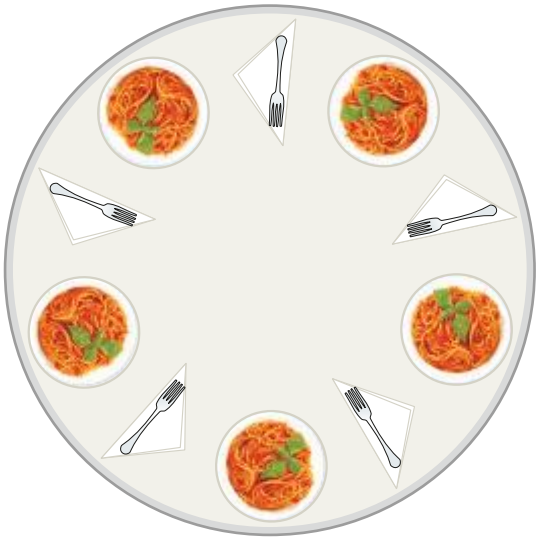
La motivazione dell'esistenza di "questa tecnica" sicuramente "poco scientifica" si basa sulla considerazione che spesso è troppo costoso mettere in atto le precauzioni sopra descritte e, basandosi su analisi statistiche, si preferisce ignorare il problema e affrontarlo poi solo nel momento in cui "in un caso remoto" potesse succedere (generalmente effettuando il reset completo di tutto il sistema).

Questa tecnica è quella utilizzata sia dal sistema operativo Unix che dalla Java Virtual Machine.

Esempio classico: problema dei filosofi a cena

Un problema classico che doverosamente deve essere ricordato, sempre dovuto a Dijkstra che lo propose alla "comunità informatica" nel 1965, è quello che viene riportato col nome di "problema dei filosofi a cena".

Una possibile formulazione è la seguente:
cinque filosofi sono seduti attorno a un tavolo circolare, ciascuno di essi ha di fronte un piatto di spaghetti che necessita di due forchette per poter essere mangiato e sul tavolo vi sono in totale solo cinque forchette disposte come nel disegno:



Ogni filosofo ha un comportamento ripetitivo, che alterna due fasi:
- una fase in cui pensa, lasciando le forchette sul tavolo;
- una fase in cui mangia, per la quale ha bisogno di avere in ciascuna mano una forchetta.

La prima considerazione che possiamo fare immediatamente è che i filosofi non possono mangiare tutti insieme: dato che ci sono solo cinque forchette solo due filosofi alla volta possono nutrirsi (altrimenti servirebbero dieci forchette).
La seconda è che due filosofi vicini di posto non possono mangiare contemporaneamente perché condividono una forchetta e, pertanto, quando uno mangia, l'altro è costretto ad attendere che i suoi vicini finiscano per potersi impossessare delle risorse.

Vediamo un esempio di funzionamento, supponendo che il filosofo quando ha fame e smette di pensare si comporti nel seguente modo:

1. come prima mossa prende la forchetta a sinistra del suo piatto;
2. quindi prende quella che è alla destra del suo piatto;
3. mangia finché è sazio;
4. quindi rimette a posto, sul tavolo, le due forchette.

Cosa succede se contemporaneamente i cinque filosofi prendono la forchetta di sinistra?
I filosofi muoiono di fame, in quanto il sistema andrebbe in deadlock: ciascun filosofo rimarrebbe con una sola forchetta in mano in attesa di un evento che non si potrà mai verificare!

Una soluzione è quella precedentemente descritta della "allocazione totale": ogni filosofo verifica se entrambe le forchette sono disponibili e solo in questo caso le acquisisce contemporaneamente, altrimenti rimane in attesa; in questo modo non si può verificare deadlock in quanto è stata rimossa la condizione di hold & wait.

SPAGHETTI O RISO?

La versione del problema che è stata sopra riportata è quella originale dall'autore: la leggenda narra che durante un soggiorno in Italia, successivo alla sua pubblicazione, a Dijkstra venne insegnato a mangiare gli spaghetti con una sola forchetta e, quindi, la versione fu modificata introducendo filosofi orientali con piatti di riso al posto degli spaghetti da mangiare con le bacchette (chopstick) al posto di forchette!

Verifichiamo le conoscenze

Esercizi a scelta multipla

1 Affinché ci sia un deadlock è necessaria la presenza di (indicare quella errata):

a) mutua esclusione
b) presenza di prerilascio
c) preemption
d) richieste bloccanti
e) attesa circolare

2 Con risorsa seriale si intende:

a) risorse richieste una dopo l'altra
b) risorse disponibili in serie
c) risorse allocate una dopo l'altra
d) nessuna delle precedenti

3 Una richiesta bloccante è anche detta:

a) "hold and wait"
b) "look and wait"
c) "look and signal"
d) "keep and signal"
e) "keep and wait"
f) "wait and signal"

4 Quale delle seguenti affermazioni è falsa in merito ai grafi di Holt?

a) Il grafo di Holt contiene un ciclo se e solo se il grafo wait-for contiene un ciclo.
b) Il I Teorema sul grafo di Holt dice quando uno stato è di deadlock
c) Il II Teorema sul grafo di Holt dice quando uno stato è di deadlock
d) La presenza di un ciclo è condizione necessaria ma non sufficiente per avere un deadlock
e) La presenza di un ciclo è condizione sufficiente ma non necessaria per avere un deadlock

5 Quale tra le seguenti non è una strategia per affrontare lo stallo?

a) detection e recovery
b) avoidance
c) prevention e recovery
d) prevention

6 I concetti di stato sicuro e sequenza sicura di esecuzione si introducono nella strategia di:

a) detection e recovery
b) avoidance
c) prevention e recovery
d) prevention

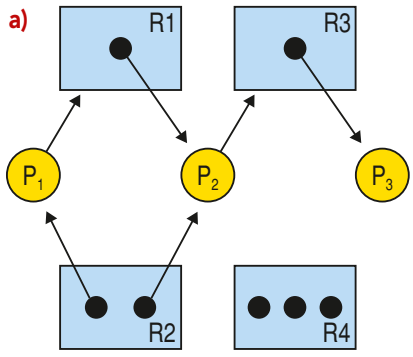
Test vero/falso

1 L'attesa circolare è condizione sufficiente al verificarsi del deadlock. V F
2 L'attesa circolare è condizione necessaria al verificarsi del deadlock. V F
3 La mutua esclusione è condizione sufficiente al verificarsi del deadlock. V F
4 Il deadlock si può verificare anche quando una sola delle condizioni necessarie è soddisfatta. V F
5 La situazione di deadlock è rappresentata dall'esistenza di un ciclo in un grafo di attesa. V F
6 Il Teorema sul grafo di Holt dice quando uno stato è di deadlock. V F
7 La terminazione totale è un metodo efficiente per eliminare lo stallo. V F
8 Nell'algoritmo del banchiere un processo può sperimentare un'attesa di durata indefinita. V F
9 Il rollback costituisce un metodo per la risoluzione del deadlock. V F
10 L'uso dei semafori garantisce il programmatore nei confronti del deadlock. V F
11 Due processi che non accedono mai a risorse comuni non possono mai essere coinvolti in una situazione di stallo. V F

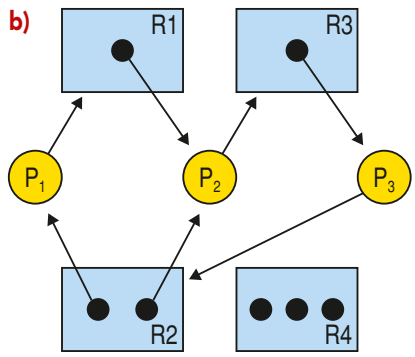
Verifichiamo le competenze

1 Nei seguenti grafi delle allocazioni è possibile che si verifichi un deadlock? Determina la risposta individuando gli eventuali cicli presenti.

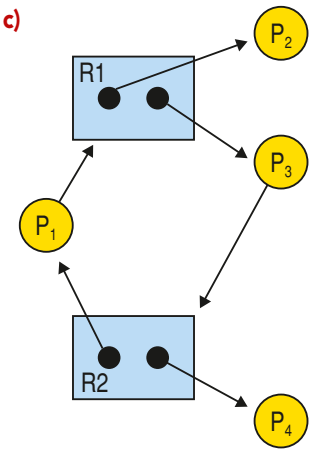
a)



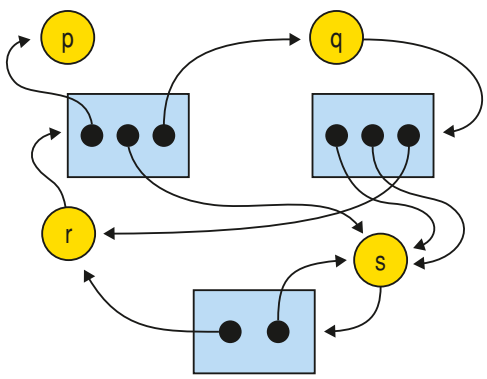
b)



c)



2 Nel seguente grafo delle allocazioni è possibile che si verifichi un deadlock? Determina la risposta effettuando le opportune riduzioni.



3 Dati tre processi A,B,C che osservano il paradigma di "possesso e attesa" e tre risorse singole Q, R, S, utilizzabili in mutua esclusione e senza possibilità di prerilascio, supponiamo che il gestore assegni le risorse al processo richiedente alla sola condizione che la risorsa sia disponibile.
Inizialmente tutte le risorse sono disponibili.
Si considerino le seguenti sequenze di richieste e rilasci:

Sequenza S1:

1) A richiede Q;
2) C richiede S;
3) C richiede Q;
4) B richiede R;
5) B richiede S;
6) A rilascia Q;
7) A richiede S;
8) C richiede R.

Sequenza S2:

1) A richiede R;
2) B richiede Q;
3) C richiede S;
4) C richiede R;
5) A richiede Q;
6) B rilascia Q;
7) B richiede S;
8) A rilascia R.

Queste sequenze provocano stallo? Motivare le risposte rappresentando nel grafo lo stato al termine di ciascuna sequenza.

4 Un sistema è dotato di 5 risorse (R,S,T,U,V) seriali, non prerilasciabili, contese da cinque processi (A,B,C,D,E): il sistema alloca le risorse disponibili al primo processo che le richiede.
La sequenza di richieste è la seguente:

A richiede R	C richiede T	B richiede R	D richiede V	E richiede R
A richiede S	A richiede T	E richiede U	C richiede U	D richiede T

Dopo l'ultima richiesta il sistema è in deadlock?
In caso affermativo dire quali sono i processi la cui terminazione forzata permetterebbe di far terminare correttamente il maggior numero di processi.
In caso negativo dire quale ulteriore richiesta porterebbe il sistema in deadlock.
Rappresentare lo stato del sistema con il grafo di Holt e motivare la risposta.

5 Algoritmo del banchiere multivaluta

Si estenda il problema del banchiere ipotizzando che questo debba fare prestiti usando valute diverse (euro, dollari, yen ecc.) e si discuta la politica di allocazione che garantisce l'assenza di deadlock.

6 Considerate i seguenti processi:

```
Risorse condivise
   semaphore S=0, M=1;
   int x=1;
Processo P1{
  while (true) do
  begin
     down(M);
     x:=2*x;
     up(M);
     up(S);
end;
}
```

```
Processo P2{
   while (true) do
   begin
      down(&);
      write(x);
      up(&);
      down(&);
end;
}
```

1) I due processi possono andare in deadlock? Motivare la risposta
2) Uno dei due processi può essere eseguito infinite volte mentre l'altro è bloccato? Motivare la risposta.
3) Descrivere tutti gli output del programma.

LEZIONE 8

I MONITOR

IN QUESTA LEZIONE IMPAREREMO...

- il concetto di monitor
- come utilizzare i monitor

Generalità

I semafori sono un meccanismo molto potente per realizzare la sincronizzazione dei processi ma il loro utilizzo è spesso molto rischioso e talvolta difficoltoso in quanto sono primitive ancora di "basso livello" e il programmatore può causare situazioni di blocco infinito (*deadlock*) o anche esecuzioni erronee di difficile verifica (*race condition*) per un errato o improprio posizionamento delle primitive di P() e V().

> Viene lasciata al programmatore troppa responsabilità nell'utilizzo di queste strutture di controllo così delicate.

Per ovviare a problemi di questa natura nei linguaggi evoluti di alto livello, come per esempio il Concurrent Pascal, Ada e Java, si sono introdotti costrutti linguistici "a più alto livello" per realizzare il controllo esplicito delle regioni critiche, dove è il compilatore che introduce il codice necessario al controllo degli accessi: questo meccanismo fu chiamato *monitor*, proposto da Hoare nel 1974 e da Brinch-Hansen nel 1975.

La definizione di monitor di Hoare è la seguente

MONITOR DI HOARE

Costrutto sintattico che associa un insieme di procedure/funzioni (public o entry) a una struttura dati comune a più processi, tale che:

- le operazioni entry sono le **sole operazioni permesse** su quella struttura;
- le operazioni entry sono **mutuamente esclusive**: un solo processo per volta può essere attivo nel monitor.

In altre parole il *monitor* definisce la regione critica e mette a disposizione sottoprogrammi che possono accedere a variabili e strutture dati interne a esso: un solo processo alla volta può essere attivo entro il *monitor* e può quindi richiamare queste procedure.
La struttura di un monitor è la seguente.

```
monitor <nome_monitor>{
   <dichiarazione delle variabili locali private>
   <inizializzazione delle variabili locali>
   /* definizione delle funzioni e procedure entry*/
   procedura entry1()          /* getter/setter delle variabili private */
      {...}
   ...
   procedura entryN()          /* getter/setter delle variabili private */
      {...}
}
```

Possiamo riconoscere in questo meccanismo il concetto di incapsulamento proprio della programmazione orientata agli oggetti, dove per accedere agli attributi privati è necessario utilizzare metodi setter/getter.

L'inizializzazione delle variabili locali viene eseguita una sola volta prima dell'esecuzione di qualunque procedura entry e tali variabili mantengono il loro valore tra successive esecuzioni delle procedure del monitor, sono cioè variabili permanenti.
A queste variabili si accede solo mediante le procedure e le funzioni entry che sono definite entro il monitor: queste procedure potrebbero anche richiamare altre procedure sempre definite dentro il monitor ma non accessibili dall'esterno, chiamate procedure non entry (sono procedure non public).

Utilizzo dei monitor

Il monitor viene utilizzato per effettuare il controllo degli accessi a una risorsa condivisa tra processi concorrenti in accordo a determinate politiche di gestione: le variabili locali definiscono lo stato della risorsa associata al monitor e i processi possono aggiornare lo stato della risorsa mediante le procedure entry.

Una istanza di tipo monitor (per esempio miaRisorsa) viene creata direttamente dopo la definizione del tipo monitor desiderato, come una qualunque variabile: nel nostro pseudocodice utilizziamo una annotazione Java-like e quindi la "manipoliamo" mediante la ◀ **dot notation** ▶.

```
monitor TipoRisorsa          /* dichiaro il tipo monitor */
{
                             <dichiarazione variabili e procedure del monitor>
}
TipoRisorsa miaRisorsa;      /* creo una istanza/oggetto di tipo monitor */
…
miaRisorsa.entry4()          /* chiamata di entry4 sull'istanza di monitor */
```

◀ **Dot notation** The popular object-oriented programming languages (most notably C++ and Java) use the industry-standard "dot notation" to refer to objects and their properties. This notation takes the form of objectname.variable. ▶

La garanzia di mutua esclusione da sola può non bastare per consentire sincronizzazione intelligente: infatti l'accesso, e quindi l'assegnazione della risorsa, avviene secondo due livelli di controllo:

1. il primo livello è quello che garantisce la mutua esclusione facendo in modo che un solo processo alla volta possa eseguire le entry (funzioni pubblic) e quindi avere accesso alle variabili comuni del monitor: nel caso di richieste contemporanee effettuate da più processi mentre uno di essi è "dentro il monitor", gli altri vengono sospesi e messi nella entry queue.
2. il secondo livello controlla l'ordine con il quale i processi hanno accesso alla risorsa: alla chiamata della procedura, se non viene verificata una condizione logica che assicura l'ordinamento, il processo viene sospeso, viene posto nella entry queue e viene liberato il monitor.

> I monitor devono essere presenti nel linguaggio di programmazione e sono stati progettati per sistemi con memoria comune; non funzionano in ambiente distribuito.

Variabili condizione e procedure di wait/signal

La condizione di sincronizzazione è costituita da variabili locali al monitor e da variabili proprie del processo, passate come parametri.

Nel caso in cui la condizione non sia verificata, la sospensione del processo avviene utilizzando variabili di un nuovo tipo, detto condition variables (condizione), che non sono dei semplici contatori ma rappresentano una coda nella quale i processi si sospendono.

Sulle variabili di tipo condition si accede solamente mediante due procedure, wait() e signal(), che hanno come parametro la variabile sulla quale devono operare:

Wait ()

L'invocazione dell'operazione wait() avviene mediante la notazione

```
miaCondition.wait()          // forza l'attesa del processo chiamante
```

dove miaCondition è la variabile condition che deve essere testata: il processo che la esegue si sospende e viene posto nella coda associata a tale variabile e libera il monitor.

Signal ()

L'invocazione dell'operazione signal() avviene mediante la notazione

```
miaCondition.signal()                    // risveglia il processo in attesa
```

dove miaCondition è la variabile condition dove si vuole liberare un processo in attesa: se la coda a esso associata contiene almeno un processo, questo viene risvegliato e riprenderà l'esecuzione da dentro il monitor, a partire dall'istruzione seguente dalla wait() che lo aveva sospeso.
Se non sono presenti processi in coda l'operazione non ha nessun effetto.

L'esecuzione della signal deve essere l'ultima istruzione eseguita dal processo all'interno del monitor in modo che se viene risvegliato un processo in attesa sullo stesso monitor questo lo trovi libero.

> La chiamata di una signal fa sì che almeno due processi possono evolvere contemporaneamente, quello che l'ha eseguita e quello risvegliato dalla coda: nella proposta di Hoare chi fa la signal si sospende immediatamente e va subito in esecuzione il processo appena risvegliato (signal and wait) mentre nella proposta di Hansen la scelta viene lasciata alla naturale gestione dello scheduler (signal and continue) semplicemente mettendo il processo che viene risvegliato nello stato di pronto.

Vediamo un primo esempio di utilizzo del monitor come allocatore di una risorsa.

ESEMPIO 14 *Monitor utilizzato come semaforo*

Utilizziamo il monitor come un semaforo, che permette o meno l'acceso a una risorsa.
Il codice è il seguente:

```
monitor allocatore
   boolean occupato=false          // variabile privata del monitor (semaforo)
   condition libero                // variabile condition

   procedura entry richiesta( )
   inizio
      if (occupato)                // se il semaforo è rosso
         libero.wait               // sospendo il processo in coda
      occupato=true                // metto il semaforo a rosso
   fine

   procedura entry rilascio ( )
   inizio
      occupato=false               // se il semaforo è verde
      libero.signal                // risveglio il processo in coda
   fine
fine monitor
```

Si può osservare come la variabile occupato venga utilizzata come un semaforo, la entry richiesta equivale alla esecuzione di un P() mentre la entry rilascio a quella di un V(): ma l'accesso esclusivo viene gestito ad alto livello e il programmatore non si deve preoccupare di realizzare la mutua esclusione ma semplicemente di richiamare queste entry all'interno dei suoi processi.

Vediamo un secondo esempio di utilizzo del monitor come gestore di una risorsa.

ESEMPIO 15 *Monitor per il problema dei produttori/consumatori*

Utilizziamo il monitor per risolvere il problema dei produttori e consumatori dove la risorsa condivisa è un buffer realizzato con un solo valore integer.
Per poter accedere alla variabile buffer mettiamo a disposizione due entry, metti e togli, definite all'interno del monitor.

La struttura del monitor è la seguente:

```
monitor ProduciConsuma
   condition riempito, svuotato
   integer buffer

   procedura entry metti(<dato>)
   inizio
      if buffer <> 0               // è presente un dato
         then svuotato.wait;       // si sospende nella coda
      <inserisci il dato>;         // quando è vuoto il buffer, mette il dato
      riempito.signal;             // risveglia il consumatore
   fine
```

```
   procedura entry preleva()
   inizio
      if buffer = 0                  // il buffer è vuoto
        then riempito.wait           // attendi che venga riempito
      <preleva dal contenitore>      // togli il dato dal buffer
      svuotato.signal                // risveglia il produttore
   fine
fine monitor
```

Scriviamo ora uno schema delle procedure produttore e consumatore che utilizzano il monitor appena descritto:

```
processo produttore()
inizio
   ...
   <produci dato>
   buffer.metti(<dato>)
...
fine

processo consumatore()
inizio
   ...
   <dato>=buffer.preleva()
   ...
fine
```

Svuotato è in pratica una coda dove gli scrittori attendono che il buffer venga "svuotato" dai lettori per poterlo riempire, e riempito è una coda di lettori in attesa di un dato da leggere.

Emulazione di monitor con i semafori

Nei linguaggi che non hanno il costrutto monitor è possibile realizzarlo mediante semafori, anche se non è molto semplice.

Infatti per ogni istanza di monitor il compilatore deve assegnare un semaforo mutex per realizzare la mutua esclusione all'accesso del monitor e un ulteriore semaforo semVar da associare a ogni variabile di tipo condition che viene definita all'interno del monitor da un contatore per tener conto del numero di processi sospesi su di esso.

Vediamo un esempio di codice per le funzioni di wait() e di signal() su di una variabile var:

```
wait(var)
 inizio
  contaVar++
  V(mutex)
  P(semC)
```

```
  P(mutex)
 fine

signal(var)
 inizio
  if(contaVar >0)
   then
    inizio
    contaVar--
    V(semS)
   fine
 fine
```

Il codice descritto è quello della proposta di Hansen, cioè signal and continue: la codifica per la proposta di Hoare, cioè signal and wait, è la seguente:

```
wait(var)
 inizio
  contaVar++
  V(mutex)
  P(semC)
 fine

signal(var)
 inizio
  if(contaVar >0)
   then
    inizio
    contaVar--
    V(semS)
    P(mutex)
   fine
 fine
```

Verifichiamo le competenze

1 Utilizzando i monitor realizza un meccanismo di sincronizzazione in modo tale che la risorsa condivisa venga assegnata a quello tra tutti i processi sospesi che la userà per il periodo di tempo inferiore.

2 Descrivi, utilizzando i monitor, il problema produttore/consumatore dove la variabile condivisa è un buffer circolare, a partire dal seguente schema di monitor:

```
Monitor buffer {
condition vuoto,pieno;
int buffer[DIM];
int contatore=0;
int preleva=0, inserisci=0;

entry riempi(int valore) {
. . .
}

entry svuota() {
. . .
}
fine monitor
```

3 Mediante un monitor realizza la seguente politica di allocazione della risorsa:

1) un nuovo lettore non può acquisire la risorsa se c'è uno scrittore in attesa;
2) tutti i lettori sospesi al termine di una scrittura hanno priorità sul successivo scrittore.

Utilizza il seguente schema di monitor:

```
monitor lettori_scrittori
{
   int num-lettori=0,occupato=0;
   condition ok_lettura,ok_scrittura;

   entry inizio_lettura()
   . . .
   }
   entry fine_lettura()
   . . .
   }
   entry inizio_scrittura()
   . . .
   }
   entry fine_scrittura()
   . . .
   }
fine monitor
```

LEZIONE 9

LO SCAMBIO DI MESSAGGI

IN QUESTA LEZIONE IMPAREREMO...

- le tipologie dei meccanismi a scambio di messaggi
- le primitive send() e receive()

Generalità

Nel modello ad ambiente locale ogni processo può accedere esclusivamente alle risorse allocate nella propria memoria (virtuale) locale che non può essere modificata direttamente dagli altri processi.

Non avendo una memoria condivisa, i processi non possono utilizzare la memoria per il coordinamento delle loro attività e lo strumento di comunicazione e sincronizzazione diventa lo *scambio di messaggi*.

In questo modello ogni processo può accedere esclusivamente alle risorse allocate nella propria memoria (virtuale) locale e ogni risorsa del sistema è accessibile a un solo processo che può operare su di essa (processo server), eseguire le operazioni che gli vengono richieste dai processi applicativi (processi clienti) che ne hanno bisogno.

> In questo modello architetturale di macchina concorrente il concetto di gestore di una risorsa coincide con quello di processo server.

Client e server comunicano tra loro esclusivamente interagendo con un meccanismo di scambio di messaggi.

Esistono due possibili modalità per implementare questo modello:

- **A** utilizzare linguaggi che prevedono costrutti espliciti per realizzare lo scambio di messaggi, come il CSP (Communicating Sequential Processes) proposto da Tony Hoare;
- **B** utilizzare la "chiamata di procedura remota", come il DP (Distributed Processes), proposto da Brinch Hansen.

In questa lezione descriveremo per ciascuno le caratteristiche essenziali e le primitive che ne permettono il funzionamento.

Canali di comunicazione

Prima di analizzare le modalità e le primitive che permettono di scambiare messaggi a due processi remoti di scambio, è necessario definire il collegamento logico mediante il quale due processi comunicano, che prende il nome di canale.

Per poter inviare un messaggio un processo deve indicare quale canale il sistema operativo deve utilizzare per effettuare la trasmissione, quindi è compito del nucleo del sistema operativo fornire al processo come primitiva l'astrazione del concetto di canale, che, nel caso di un sistema distribuito, sarà la rete di comunicazione (per esempio Internet), mentre nel caso di un sistema multiprocessori può essere un bus locale.

Il programmatore non deve curarsi dell'hardware e quindi deve avere a disposizione costrutti linguistici ad alto livello che gli permettano di definire logicamente i canali, e linguaggi di programmazione che gli offrano le primitive che gli permettono di utilizzarli per programmare le varie interazioni tra i processi della sua specifica applicazione.

Un canale è caratterizzato da tre parametri:

A la tipologia del canale, intesa come direzione del flusso dei dati che un canale può trasferire, che può essere:
- monodirezionale: il flusso di dati viene trasmesso in una sola direzione;
- bidirezionale: viene usato sia per inviare che per ricevere informazioni.

B la designazione del canale e dei processi sorgente e destinatario di ogni comunicazione, che può essere:
- nel caso di comunicazione asimmetrica il mittente nomina esplicitamente il destinatario ma questo non nomina esplicitamente il mittente:
  - port: canale asimmetrico da-molti-a-uno, i processi clienti specificano il destinatario delle loro richieste e il processo servitore è pronto a ricevere messaggi da qualunque cliente;
  - mailbox: canale asimmetrico da-molti-a-molti processi cliente che inviano richieste non a un particolare servitore, ma a uno qualunque scelto tra un insieme di servitori equivalenti;
- nel caso di comunicazione simmetrica entrambi si nominano in modo esplicito:
  - link: canale simmetrico da-uno-a-uno;

C il tipo di sincronizzazione fra i processi comunicanti.
- comunicazione asincrona;
- comunicazione sincrona:
  - semplice o stretta;
  - estesa.

Comunicazione asincrona

Nel caso di comunicazione asincrona la comunicazione da parte del processo mittente avviene senza che questo rimanga in attesa di una risposta da parte del processo destinatario, quindi il processo mittente continua la sua esecuzione immediatamente dopo che il messaggio è stato inviato.

> La spedizione del messaggio non è un punto di sincronizzazione, è quindi necessario utilizzare questo schema di interazione quando lo scambio esplicito di messaggi non deve contenere informazioni da elaborare, in quanto è complesso verificarne l'esito e comportarsi di conseguenza.

Comunicazione sincrona

Nel caso di comunicazione sincrona lo scambio di informazioni può avvenire solo se mittente e destinatario sono pronti a "parlarsi" e quindi è necessario che si sincronizzino, e questa interazione prende il nome di "rendez-vous":

Ⓐ rendez-vous semplice o stretto
Si limita alla trasmissione di un messaggio dal mittente al destinatario: il primo dei due processi comunicanti che esegue l'invio oppure è in necessità di ricevere un dato si sospende in attesa che l'altro sia pronto a eseguire l'operazione attesa.
Il messaggio che viene inviato/ricevuto contiene informazioni corrispondenti allo stato attuale del processo mittente e questo semplifica la scrittura dei programmi semplificando la realizzazione di meccanismi di sincronizzazione: questo modello è il CSP (Communicating Sequential Processes) proposto da Tony Hoare;

Ⓑ rendez-vous esteso
In questo caso il destinatario, una volta ricevuto il messaggio, deve inviare una risposta al mittente e quindi il processo mittente rimane in attesa fino a che il ricevente non ha terminato di svolgere l'azione richiesta.
Viene realizzato nel modello client-server mediante RPC, cioè chiamata a procedura remota; in questo caso il parallelismo risulta essere ridotto ma la sincronizzazione tra i processi è palese e quindi risulta semplice la verifica dei programmi.
Utilizza questo meccanismo il DP (Distributed Processes), proposto da Brinch Hansen.

Primitive di comunicazione asimmetrica da-molti-a-uno

Per dichiarare un canale possiamo per esempio utilizzare la seguente notazione sintattica:

```
port <tipo> <identificatore>;
```

Per esempio, se vogliamo definire un canale per trasferire messaggi che sono semplicemente numeri interi scriviamo:

```
port int canale1;
```

Con la parola riservata port si identifica un canale asimmetrico da-molti-a-uno nell' host ricevente: gli host mittenti, per inviare a questo una messaggio, utilizzano la dot notation come di seguito riportato:

```
<nome del processo>.<identificatore del canale>
```

Se sull'host dove si è definito il canale canale1 si vuole inviare un messaggio al processo processo12, si scrive semplicemente:

```
processo12.canale1;
```

Primitiva di invio: send()

Per inviare un messaggio si utilizza la primitiva send() con la seguente notazione sintattica:

```
send(<valore>) to <porta>;
```

dove
- <porta>: identifica in modo univoco il canale a cui inviare il messaggio;
- <valore>: identifica il contenuto del messaggio, che deve essere dello stesso tipo definito dalla <porta>.

ESEMPIO 16 ***Invio di un messaggio***

Trasmettiamo come messaggio il numero 666 al processo12 tramite il canale1 dal quale il destinatario può ricevere.

```
send(666) to processo12.canale1;
```

La primitiva send() non è bloccante nelle comunicazione asincrona mentre per realizzare scambi di messaggi sincroni è necessario che sia bloccante, cioè il processo che l'ha eseguita si sospenda in attesa di una risposta da parte del destinatario.

Primitiva di ricezione: receive()

Per ricevere un messaggio si utilizza la primitiva receive () con la seguente notazione sintattica:

```
receive(<variabile>)from <porta>;
```

dove
- <porta>: identifica in modo univoco la porta dell'host ricevente sulla quale si attende il messaggio;
- <variabile>: è l'identificatore della variabile dove verrà posto il messaggio appena ricevuto, che deve essere dello stesso tipo definito dalla <porta>.

ESEMPIO 17 ***Ricezione di un messaggio***

Un processo è in attesa sul canale1 di un messaggio di tipo intero che verrà memorizzato nella variabile dato:

```
receive (dato) from canale1;
```

Se non ci sono messaggi sul canale questa primitiva sospende il processo che l'ha chiamata, altrimenti preleva il primo messaggio dal canale e lo assegna alla variabile dato e inoltre memorizza in una apposita variabile un valore del tipo predefinito process che identifica il nome del processo mittente.

Vediamo un esempio completo di produttore/consumatore sincrono realizzato mediante le primitive appena definite.

```
Processo consumatore1
inizio
port int canale1;
. . .
 receive (dato) from canale1;
. . .

fine
```

```
Processo produttore1
inizio
int messaggio
. . .
<produci messaggio>
. . .
send(messaggio) to consumatore1.canale1;
. . .
fine
```

La primitiva, se non ci sono messaggi sul canale bloccante, costituisce un punto di sincronizzazione, ma introduce il problema dell'attesa attiva in quanto il processo ricevente rimane in loop in attesa che arrivi il messaggio.

Un problema può sorgere nel caso di un host in attesa di ricezione di diverse richieste di servizio e tra di queste deve privilegiarne alcune rispetto alle altre: è necessario introdurre più canali di ingresso, ciascuno dedicato a un tipo di richiesta diverso, e avere a disposizione la primitiva che permetta di indicare su quali canali attendere, per esempio in base dello **stato interno** della risorsa che il processo sta gestendo.

Il meccanismo di ricezione ideale è il seguente:
- deve consentire al processo server di verificare la disponibilità di messaggi su un insieme di canali potendo anche assegnare a essi una priorità di scelta di ricezione;
- deve poter abilitare la ricezione di un messaggio da un qualunque canale non appena questo divenga disponibile, e nel caso di più messaggi pronti in contemporanea, lasciare al processo la scelta di quale privilegiare, in base alle proprie politiche di gestione;
- nel caso che nessun canale presenti messaggi disponibili deve bloccare il processo in attesa che arrivi un messaggio su di un canale qualsiasi.

Primitive di comunicazione asimmetrica da-molti-a-molti (cenni)

In questa situazione abbiamo molti processi che possono inviare messaggi a più destinatari e per realizzarla è necessario utilizzare una mailbox al posto dei canali di comunicazione.
Le primitive possono essere così modificate:
per inviare un messaggio si utilizza la primitiva send() con la seguente notazione sintattica:

```
send(<mailbox>,<messaggio>)
```

per ricevere un messaggio si utilizza la primitiva receive () con la seguente notazione sintattica:

```
receive (<mailbox>,<messaggio>)
```

dove
- `<mailbox>` indica il nome dell'area di memoria destinata a contenere i messaggi;
- `<messaggio>` indica il dato che viene trasmesso.

La realizzazione dei programmi client/server e la modalità di gestione delle porte sono oggetto di studio nel quinto anno di corso e saranno descritte nel terzo volume di questa collana.

Verifichiamo le conoscenze

Esercizi a completamento

1 Esistono due possibili modalità per implementare questo modello:

a) utilizzare linguaggi che prevedono costrutti espliciti per realizzare lo ....., come il ..... proposto da Tony Hoare;

b) utilizzare la ..... come il ..... proposto da Brinch Hansen.

2 Un canale è caratterizzato da tre parametri:

a) la **tipologia del canale**, intesa come direzione del flusso dei dati che un canale può trasferire, che può essere:

- .....: il flusso di dati viene trasmesso in una sola direzione;
- .....: viene usato sia per inviare che per ricevere informazioni.

b) la **designazione del canale** e dei processi ..... e ..... di ogni comunicazione, che può essere:

- nel caso di comunicazione ..... il mittente nomina esplicitamente il destinatario ma questo non nomina esplicitamente il mittente:
  - port: canale ..... da ..... a ....., i processi clienti specificano il **destinatario** delle loro richieste e il processo servitore è pronto a ricevere messaggi da **qualunque cliente**;
  - mailbox: canale ..... da ..... a ..... processi cliente che inviano richieste non a un particolare servitore, ma a **uno qualunque** scelto tra un insieme di **servitori equivalenti**;
- nel caso di comunicazione ..... entrambi si nominano in modo esplicito.
  - link: canale ..... da ..... a .....

b) **il tipo di sincronizzazione** fra i processi comunicanti.

- comunicazione .....;
- comunicazione .....:
  - semplice o .....;
  - .....

Domande a risposta aperta

1 Che cos'è un canale?
2 Come avviene la comunicazione asincrona?
3 Come avviene la comunicazione sincrona?
4 Che cosa si intende per rendez-vous stretto?
5 Che cosa si intende per rendez-vous esteso?
6 Quali primitive vengono utilizzate per realizzare la comunicazione mediante lo scambio di messaggi?
7 Che cosa si intende per RPC?
8 In che cosa consiste la proposta CSP di Hoare?
9 In che cosa consiste il meccanismo DP proposto da Brinch Hansen?

Esercitazioni di Laboratorio 1

I SEMAFORI IN C

Info

I codici seguenti sono reperibili nel file C_semafori_rar scaricabile dalla cartella materiali nella sezione del sito www.hoepliscuola.it riservata al presente volume.

Realizzazione dei semafori

Il linguaggio C ha nella libreria <pthread.h> il concetto di semaforo MUTEX (Mutual Exclusion), un semaforo binario che ha valore zero se è occupato (rosso) e uno se è libero (verde).

Per prima cosa dobbiamo creare una variabile semaforo:

```
pthread_mutex_t semaforo;
```

è una variabile di tipo semaforo a cui sarà associato un valore (0,1) e una coda di thread sospesi sul mutex.

Il semaforo deve essere inizializzato prima di essere usato ed è possibile farlo.

Staticamente

Al momento della creazione viene utilizzato il seguente codice per settarlo occupato, altrimenti, di default il semaforo è settato come libero:

```
pthread_mutex_t semaforo = PTHREAD_MUTEX_INITIALIZER;
```

Il significato cambia a seconda della opzione di inizializzazione, ma delle diverse possibilità noi utilizzeremo solo quella sopra indicata.

Zoom su...

INIZIALIZZAZIONE STATICA DI UN MUTEX

Le possibili forme di inizializzazione statica di un mutex sono:

- PTHREAD_MUTEX_INIZIALIZER: il semaforo parte dallo stato occupato
- PTHREAD_RECURSIVE_MUTEX_INIZIALIZER_NP: crea un semaforo binario ricorsivo: ripete il tentativo di accesso alla sezione critica per 'n' volte, per cui poi dovremo fare l'unlock 'n' volte
- PTHREAD_ERRORCHECK_MUTEX_INIZIALIZER_NP: se il semaforo è bloccato, ritorna il codice di errore EDEADLK
- PTHREAD_ADAPTIVE_MUTEX_INITIALIZER_NP: per i "fast mutex"

Dinamicamente

È disponibile la seguente funzione di libreria che consente di inizializzarlo dinamicamente:

```
pthread_mutex_init (pthread_mutex_t *semaforo, pthread_mutex_attr *attributi)
```

La funzione, come parametri, necessita di un puntatore a una variabile di tipo semaforo e gli attributi per l'inizializzazione del semaforo (NULL per i valori di default).

ESEMPIO 18 ***Inizializzazione dinamica di un mutex***

La seguente istruzione inizializza il semaforo libero

```
pthread_mutex_init(&bufferVuoto, NULL);
```

Funzioni P() e V()

Le funzioni che permettono di sospendere il thread in attesa che la risorsa associata al semaforo sia libera oppure per liberare il semaforo sono le seguenti:

```
int pthread_mutex_lock (pthread_mutex_t *semaforo)          // P(S) o wait(S)
```

se il semaforo è occupato (ovvero è *zero*) il thread viene messo nella coda di attesa a esso associata, altrimenti occupa direttamente il semaforo e ritorna 0.

Esiste anche una seconda istruzione di lock(), la *trylock() che* non è bloccante.

```
int pthread_mutex_trylock (pthread_mutex_t *semaforo)   // P(S) o wait(S)
```

Se il mutex è occupato trylock restituisce EBUSY

Per rilasciare la sezione critica e quindi liberare il semaforo si utilizza la funzione

```
int pthread_mutex_unlock (pthread_mutex_t *semaforo)
```

che provvede a porre il semaforo al valore uno (cioè verde) oppure, se ci sono processi in coda, ne risveglia il primo.

Soluzione del problema del produttore-consumatore

Vediamo un primo esempio dove sincronizziamo due processi, uno produttore e uno consumatore, utilizzando due semafori, secondo il seguente schema generale:

```
semaforo bufferVuoto = verde;
semaforo datoPronto = rosso;
produttore {
   P(bufferVuoto);
   ... produci ...
   V(datoPronto)
```

```
}
consumatore {
   P(datoPronto);
   ... consuma ...
   V(bufferVuoto);
}
```

Facciamo eseguire dieci volte la produzione e il consumo del dato, che si limita a un numero intero incrementato volta per volta dal produttore e inserito in una variabile condivisa.
Dato che il produttore testa il semaforo verde, sarà il primo ad accedere alla regione condivisa e a depositare un dato: finito di scrivere sveglierà il consumatore che inizierà a leggerlo e solo a fine lettura metterà a verde il semaforo di buffer vuoto, permettendo al produttore di inserire un nuovo dato… e così via per dieci iterazioni.

La codifica in C richiede l'importazione di tre librerie e la definizione dei due semafori, dei quali quello che indica il dato pronto viene inizializzato staticamente a occupato:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int variabileCondivisa;                 /* area di memoria condivisa          */
pthread_mutex_t bufferVuoto;            /* semaforo verde se buffer vuoto     */
                                        /* semaforo verde se buffer pieno     */
pthread_mutex_t datoPronto = PTHREAD_MUTEX_INITIALIZER; /* all'inizio è vuoto*/
```

Il codice dei thread produttore e consumatore non richiede spiegazioni aggiuntive. ▶

```c
void *produttore (void *parametro) {    /*codice del processo produttore     */
  int conta = 0;
  while (conta<10) {
   pthread_mutex_lock(&bufferVuoto);    /* P(bufferVuoto)aspetta buffer vuoto*/
   variabileCondivisa = ++conta;        /* produci un numero progressivo      */
   printf("Prodotto: %d\n", variabileCondivisa);
   pthread_mutex_unlock(&datoPronto);   /* V(datoPronto) per il consumatore   */
  }
  pthread_exit(0);
}

void *consumatore (void *parametro) {   /*codice del processo consumatore     */
 int conta = 0;
 while (conta<10) {
  pthread_mutex_lock(&datoPronto);      /* P(datoPronto) aspetta il dato      */
  conta = variabileCondivisa;           /* consuma il numero progressivo      */
  printf("Consumato: %d\n", conta);
  pthread_mutex_unlock(&bufferVuoto);   /* V(bufferVuoto) per il produttore   */
 }
 pthread_exit(0);
}
```

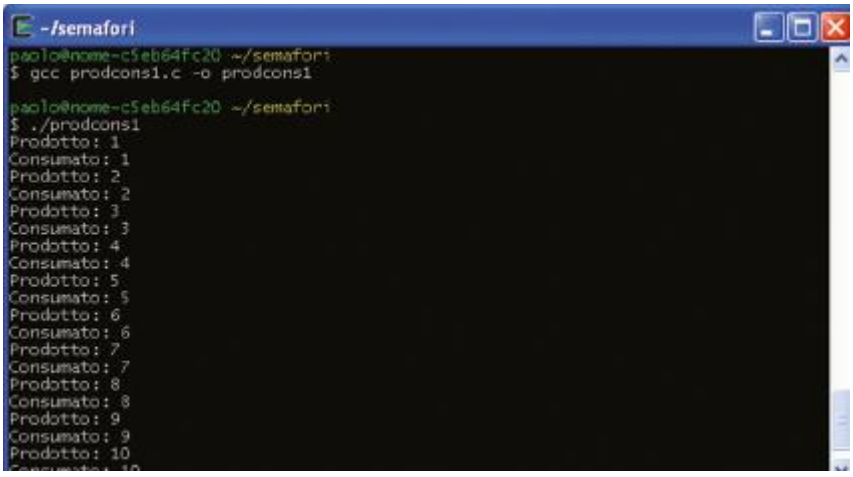
Il codice del main è il seguente: ▶

```c
int main() {
  pthread_t produci;       /*dichiaro le variabili relative ai thread*/
  pthread_t consuma;

  pthread_mutex_init(&bufferVuoto,NULL); /*inizializzo il semaforo a libero   */
  datoPronto =0;                         /*inizializzo il semaforo a occupato */

  pthread_create(&produci,NULL,produttore,NULL);          /*creo il produttore*/
  pthread_create(&consuma,NULL,consumatore,NULL);         /*creo il consumatore*/

  pthread_join(produci,NULL);             /*attesa della terminazione dei figli*/
  pthread_join(consuma,NULL);

  pthread_mutex_destroy(&bufferVuoto);              /*distruzione dei semafori*/
  pthread_mutex_destroy(&datoPronto);
  return 0;
}
```

Dopo aver dichiarato i due thread, vengono inizializzati i semafori, ma l'istruzione 38 è superflua in quanto il semaforo datoPronto è già stato inizializzato staticamente con l'istruzione 8.

Le istruzioni 46 e 47 servono per liberare memoria dato che i semafori ora non servono più.

Compiliamo il tutto e mandiamolo in esecuzione, ottenendo il seguente output:



Prova adesso!

1. Modifica il programma precedente in modo che siano presenti due produttori e due consumatori: il primo produttore incrementa il dato di una sola unità mentre il secondo lo raddoppia. A video i consumatori visualizzano oltre al valore letto anche il proprio nome.
2. Realizza un programma per sincronizzare un produttore che riempie una coda circolare di 10 elementi e tre consumatori che estraggono dalla coda un elemento ciascuno.
   Il produttore incrementa un contatore a ogni scrittura e il consumatore lo visualizza sullo schermo, assieme al proprio identificatore.
3. Successivamente realizza una situazione dove N produttori e M consumatori utilizzano una risorsa con memoria limitata, per esempio composta da 10 elementi.
   Manda in esecuzione una possibile situazione di tre produttori e tre consumatori che rispettivamente inseriscono un numero progressivo e lo leggono dal buffer visualizzandolo sullo schermo.
4. Realizza un programma per sincronizzare tre processi che rispettivamente visualizzano il suono di una campana in modo da ottenere la sequenza infinita di "DIN" "DON" "DAN".
5. Realizza un programma che sincronizzi la situazione lettori/scrittori, dove più lettori possono contemporaneamente accedere alla risorsa condivisa mentre gli scrittori devono alternarsi, nei seguenti casi:
   - A privilegiando i lettori
   - B privilegiando gli scrittori
   - C alternando il più possibile lettori e scrittori

Esercitazioni di Laboratorio 2

I MONITOR IN C

Pthread condition variable

Il linguaggio C ha nella libreria <pthread.h> oltre al semaforo MUTEX (Mutual Exclusion) la condition variable che viene utilizzata per sospendere l'esecuzione di un thread in attesa che si verifichi un certo evento.

> Le condition variable vanno sempre utilizzate associandole a un mutex per evitare che si verifichino problemi di deadlock.

La dichiarazione di una condition variable avviene con la seguente istruzione:

```
pthread_cond_t <nome;
```

E, come per il mutex, viene inizializzata tramite la seguente sintassi:

```
pthread_cond_t <nome>=PTHREAD_COND_INITIALIZER;
```

Per mettersi in attesa un thread utilizza la seguente system call:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

che richiede come parametri due semafori, il semaforo condizione sul quale eventualmente sospendersi e il mutex che regola la mutua esclusione alla regione critica.

La funzione che risveglia i thread sospesi viene fatta tramite la seguente system call

```
int pthread_cond_signal(pthread_cond_t *cond);
```

> Questa istruzione risveglia il primo thread sospeso: ne esiste anche una che risveglia tutti i thread in attesa sulla condizione cond:
>
> ```
> int pthread_cond_broadcast (pthread_cond_t *cond);
> ```

Sincronizzazione con condition variable

Per realizzare la sincronizzazione di thread con variabili condizione per prima cosa le definiamo e inizializziamo:

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t libero=PTHREAD_COND_INITIALIZER;
```

Lo schema della procedura/funzione di un thread che si metta in attesa sulla variabile condition è il seguente:

```
pthread_mutex_lock(&mutex);
...
pthread_cond_wait(&libero,&mutex );
...
pthread_mutex_unlock(&mutex);
```

Lo schema della procedura/funzione di un thread che libera la situazione di attesa sulla variabile condition è il seguente:

```
pthread_mutex_lock(&mutex);
...
pthread_cond_signal(&libero,&mutex );
...
pthread_mutex_unlock(&mutex);
```

Vediamo un esempio completo di codice dove un primo thread si sospende su un semaforo in attesa che un secondo thread lo "risvegli" al termine della sua elaborazione.

Definiamo due semafori, un mutex e una condition variable:

```
1 #include <stdio.h>
2 #include <pthread.h>
3 pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;      /* mutex */
4 pthread_cond_t condVar=PTHREAD_COND_INITIALIZER;      /* condition variable */
```

Il codice dei due thread rispecchia lo schema prima descritto:

```
6 void thread1_func(void *ptr){
7   printf("Avvio esecuzione del %s.\n",(char *)ptr);
8   sleep(2); /* pausa di 2 secondi */
9   printf("Thread 1 prova ad entrare nella sezione critica.\n");
10  pthread_mutex_lock(&mutex);
11  printf("Thread 1 nella sezione critica e si sospende sulla condVar.\n");
12  pthread_cond_wait(&condVar, &mutex);
13  printf("Thread 1 viene risvegliato dal Thread 2 e lascia la sezione critica\n");
14  pthread_mutex_unlock(&mutex);
15  printf("Thread 1 termina elaborazione.\n");
16 }
```

Appena avviata l'esecuzione, aspetta due secondi prima di entrare nella sezione critica: quindi effettua il test sulla condition variable condVar e, trovandola a valore falso, si sospende in attesa che questa venga modificata da qualche altro thread.

```
17 void thread2_func(void *ptr) {
18   printf("Avvio esecuzione del %s.\n",(char *)ptr);
19   sleep(5); /* pausa di 5 secondi */
20   printf(" Thread 2 entra nella sezione critica.\n");
21   pthread_mutex_lock(&mutex);
22   printf(" Thread 2 risveglia il Thread 1 in attesa sulla condition variable.\n");
23   pthread_cond_signal(&condVar);
24   printf(" Thread 2 esce dalla sezione critica .\n");
25   pthread_mutex_unlock(&mutex);
26   printf(" Thread 2 termina elaborazione.\n");
```

Il secondo thread aspetta cinque secondi prima di entrare nella sezione critica e dentro essa modifica il valore della condition variable condVar in modo da risvegliare il primo thread che era sospeso su di essa, quindi termina l'elaborazione.

Un'esecuzione è la seguente:

```
~/semafori
paolo@nome-c5eb64fc20 ~/semafori
$ gcc sincroVC1.c -o sincroVC1

paolo@nome-c5eb64fc20 ~/semafori
$ ./sincroVC1
Avvio esecuzione del Thread 1.
Avvio esecuzione del Thread 2.
Thread 1 prova ad entrare nella sezione critica.
Thread 1 nella sezione critica e si sospende sulla condVar.
 Thread 2 entra nella sezione critica.
 Thread 2 risveglia il Thread 1 in attesa sulla condition variable.
 Thread 2 esce dalla sezione critica .
 Thread 2 termina elaborazione.
Thread 1 viene risvegliato dal Thread 2 e lascia la sezione critica
Thread 1 termina elaborazione.

paolo@nome-c5eb64fc20 ~/semafori
$
```

dove il programma main si limita a creare i due thread e a mandarli in esecuzione:

```
30 int main() {
31   pthread_t thread1,thread2;
32   char *msg1="Thread 1";
33   char *msg2="Thread 2";
34   if(pthread_create(&thread1,NULL,(void *)&thread1_func,(void *)msg1)!=0) {
35    perror("Errore nella creazione del primo thread.\n");
36    return 1;
37   }
38   if(pthread_create(&thread2,NULL,(void *)&thread2_func,(void *)msg2)!=0) {
39    perror("Errore nella creazione del secondo thread.\n");
40   return 1;
41   }
42   pthread_join(thread1,NULL);
43   pthread_join(thread2,NULL);
44 return 0;
```

Pthread monitor

Nella libreria <pthread.h> non è invece previsto il costrutto linguistico "monitor", e pertanto:

- i dati "interni al monitor" sono potenzialmente accessibili direttamente da tutti i processi;
- la mutua esclusione delle funzioni/procedure entry deve essere garantita esplicitamente dal programmatore mediante lock/unlock su un mutex associato al "monitor";
- è necessario che il programmatore effettui l'accesso al monitor esclusivamente attraverso le funzioni/procedure entry.

Risolviamo come esempio il classico problema del produttore e consumatore nel primo caso, cioè quello in cui è presente un solo produttore e un solo consumatore.

Definiamo un record contenente la regione di memoria condivisa (int buffer) e i semafori per accedere a essa: un mutex e due semafori condition, col codice seguente:

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <pthread.h>
 4
 5 typedef struct{
 6  int buffer;                    /* area di memoria condivisa              */
 7  pthread_mutex_t mutex;         /* mutex per la mutua esclusione          */
 8  pthread_cond_t pieno;          /* condition var per accedere a consumare */
 9  pthread_cond_t vuoto;          /* condition var per accedere a produrre  */
10 }prodcons;
```

Definiamo ora le primitive che possono operare su questa struttura dati: innanzi tutto scriviamo la funzione che provvede alla loro inizializzazione che sarà chiamata dal main prima della creazione dei thread (istruzione 72).

```
12 void inizializza (prodcons *miopc){       /* entry per inizializzare il buffer */
13   pthread_mutex_init(&miopc->mutex, NULL);
14   pthread_cond_init (&miopc->pieno, NULL);
15   pthread_cond_init (&miopc->vuoto, NULL);
16   miopc->buffer=0;
17 }
```

La entry inizia e termina con la gestione della mutua esclusione regolata dal mutex: quindi controlla se il buffer è vuoto per poter inserire il proprio dato e "liberare" il thread consumatore (semaforo pieno con istruzione 25): in caso contrario si sospende sul semaforo vuoto (istruzione 23):

```
19 void inserisci (prodcons *miopc, int data){       /* entry per l'inserimento */
20  pthread_mutex_lock (&miopc->mutex);     /* mutex di mutua esclusione a rosso */
21
22  if (miopc->buffer != 0)                              /* se il buffer è pieno  */
23    pthread_cond_wait (&miopc->vuoto, &miopc->mutex);   /* si pone in attesa    */
24  miopc->buffer = data;                                /* scrivi il dato       */
25  pthread_cond_signal (&miopc->pieno);    /* risveglia eventuali consumatori    */
26
27  pthread_mutex_unlock (&miopc->mutex);   /* mutex di mutua esclusione a verde */
28 }
```

Anche questa entry inizia e termina con la gestione della mutua esclusione regolata dal mutex: quindi controlla se il buffer è vuoto e se nessun dato è stato prodotto per mettersi in attesa sul corrispondente semaforo pieno (istruzione 35): se il dato è disponibile, lo preleva, svuota il buffer e segnala al produttore che può riprendere l'attività (istruzione 38):

```
30 int estrai (prodcons *miopc){                    /* entry per l'estrazione    */
31  int data;
32  pthread_mutex_lock (&miopc->mutex);     /* mutex di mutua esclusione a rosso */
33
34  if (miopc->buffer==0)                               /* se il buffer è pieno  */
35    pthread_cond_wait (&miopc->pieno, &miopc->mutex); /* si pone in attesa     */
36  data = miopc->buffer;                               /* leggi il dato         */
37  miopc->buffer = 0;                                  /* svuota il buffer      */
38  pthread_cond_signal  (&miopc->vuoto);   /* risveglia eventuali produttori    */
39
40  pthread_mutex_unlock (&miopc->mutex);   /* mutex di mutua esclusione a verde */
41 return data;
42 }
```

Quello che abbiamo realizzato sino a ora è proprio il monitor, con la sua area dati e le primitive (entry) che permettono ai thread di potervi accedere.

Queste due entry vengono chiamate rispettivamente dai processi produttore e consumatore che, come si può vedere dai codici seguenti, non si preoccupano della sincronizzazione ma si limitano a utilizzare le funzioni messe a disposizione dal monitor.

```
44 #define MAX 10                  /* numero di dati da produrre                    */
45 #define FINE (-1)               /* valore del dato di fine produzione (tappo)    */
46 prodcons bufferComune;          /* dichiarazione della variabile comune          */
47
48 void *produttore (void *data){
49  int n;
50  printf("sono il thread produttore\n\n");
51  for (n = 1; n < MAX; n++){
52   printf ("Thread produttore: dato %d \n", n);
53   inserisci (&bufferComune, n);
54  }
55  inserisci (&bufferComune, FINE);
56  return NULL;
57 }
58
59 void *consumatore (void *data){
60  int dato;
61  do{
62   dato = estrai (&bufferComune);
63   if (dato != FINE)
64    printf("Thread consumatore: dato %d\n", dato);
65   }while (dato != FINE);
66  return NULL;
67 }
```

Anche il main non si cura della sincronizzazione, ma si limita a definire i due thread, a mandarli in esecuzione e ad attendere la loro terminazione.

```
69 main (){
70  pthread_t uno, due;
71  void *retval;
72  inizializza (&bufferComune);
73
74  pthread_create (&uno, NULL, produttore, 0);  /*  creazione di due threads  */
75  pthread_create (&due, NULL, consumatore, 0);
76
77  pthread_join (uno, &retval);                 /* attesa teminazione threads */
78  pthread_join (due, &retval);
79  return 0;
80 }
```

Prova adesso!

1. Modifica l'esempio precedente per generalizzare il problema nel caso che sia presente un buffer circolare con dimensione BUFFER_SIZE (per esempio #define BUFFER_SIZE 12). Utilizza due variabili (`int readpos, writepos;`) per leggere/scrivere nel buffer:
   - `writepos` è il puntatore alla prima posizione libera.
   - `readpos` è il puntatore al primo elemento occupato.

I thread produttori e consumatori necessitano di sincronizzazione in caso di:
- buffer pieno: definiscono una condition per la sospensione dei produttori se il buffer è pieno (`nonPieno`)
- buffer vuoto: definiscono una condition per la sospensione dei produttori se il buffer è vuoto (`notVuoto`)

La struttura dati incapsulati dal "monitor" è quindi la seguente:

```
5 #define BUFFER_SIZE 12
6 typedef struct{
7  int buffer[BUFFER_SIZE];
8  pthread_mutex_t mutex;
9  int readpos, writepos;
10 int contaDati;
11 pthread_cond_t nonVuoto;
12 pthread_cond_t notPieno;
13 }prodcons;
```

Definisci tre entry per effettuare le operazioni sul "monitor" prodcons:
- `inizializza`: inizializzazione del buffer;
- `inserisci`: operazione "entry" eseguita da ogni produttore per l'inserimento di un nuovo elemento;
- `estrai`: operazione "entry" eseguita da ogni consumatore per l'estrazione di un elemento dal buffer.

Per testare il monitor puoi utilizzare gli stessi processi definiti nel caso precedente.

2 Modifica l'esempio precedente (1 solo produttore e 1 solo consumatore) per generalizzare il problema a N produttori e N consumatori che utilizzano sempre un buffer circolare con dimensione BUFFER_SIZE (per esempio #define BUFFER_SIZE 12).
Effettua le necessarie modifiche al monitor e ai processi di prova nonché al main che deve generare rispettivamente:
- 5 processi produttori e 10 processi consumatori;
- 10 processi produttori e 5 processi consumatori;
- 10 processi produttori e 10 processi consumatori.

Discuti i risultati che hai ottenuto.

3 Realizza un programma che sincronizzi la situazione lettori/scrittori, dove più lettori possono contemporaneamente accedere alla risorsa condivisa mentre gli scrittori devono alternarsi, nei seguenti casi:
- privilegiando i lettori;
- privilegiando gli scrittori;
- alternando il più possibile lettori e scrittori.

4 Realizza un'applicazione concorrente per la gestione dello studio di un veterinario, unico per cani e gatti, dove ogni utente del veterinario (cane o gatto) è rappresentato da un thread e il suo studio come una risorsa. La politica di sincronizzazione tra i processi dovrà garantire che:
- nello studio non vi siano contemporaneamente cani e gatti;
- nell'accesso allo studio, i gatti abbiamo la priorità sui cani.

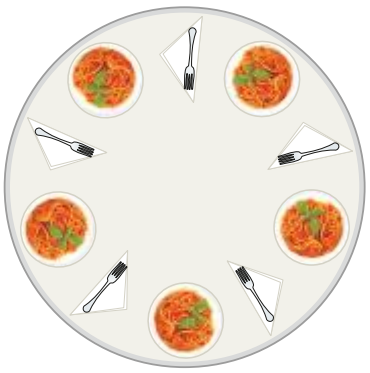
Supponi inoltre che lo studio abbia una capacità limitata a N animali. Il numero C di cani, quelli dei gatti G e la capacità N dovranno essere passati alla riga di comando.

Esercitazioni di Laboratorio 3

LA SOLUZIONE DEL DEADLOCK IL C

Filosofi e deadlock

Si vuole realizzare in linguaggio C la soluzione del problema dei 5 filosofi utilizzando i semafori della libreria <pthread.h>.



Ciascun filosofo, numerato con un valore da 0 a N (possiamo poi provare il caso di N filosofi) può compiere tre attività: pensare, mangiare e, nel caso non trovasse disponibili le forchette, aspettare. ▶

Associamo a questi tre stati dei valori costanti e indichiamo inoltre il filosofo di destra e quello di sinistra sempre con due costanti (che verranno comode nel seguito del programma): ▶

```
 5 #define N 5
 6 #define PENSA 0
 7 #define ATTESA 1
 8 #define MANGIA 2
 9 #define DESTRA    (i+1)%N
10 #define SINISTRA (i-1+N)%N
11
```

I dati in comune sono semplicemente le variabili che indicano lo stato dei filosofi e le indichiamo in un vettore: ciascuna di esse necessita di un semaforo per potervi accede in mutua esclusione:

```
11
12 pthread_mutex_t mutex, mutex_f[N];          /* semafori per RC e forchette   */
13 int stato[N];                 /* variabile condivisa con lo stato dei filosofi */
14
```

Le due attività prevalenti dei filosofi sono quelle di pensare e mangiare: vengono simulate con due funzioni:

```
15 void pensa(int i){
16  printf("\nFILOSOFO %d:  sto pensando ... ", i);
17  sleep(rand()%N);                    /* pensa per un tempo casuale tra 0 e N  */
18 }
19
20 void mangia(int i){
21  printf("\nFILOSOFO %d: sto mangiando ... ",i);
22  sleep(rand()%N);                   /* mangia per un tempo casuale tra 0 e N  */
23 }
```

Utilizzeremo lo stato dei filosofi adiacenti per sapere se il generico filoso può mangiare: se il filosofo di destra non sta mangiando e quello di sinistra non sta mangiando, sicuramente le forchette sono libere e quindi il filosofo corrente può mangiare.
Utilizziamo quindi il semaforo associato al filosofo a tale scopo, cioè bloccandolo quando inizia a mangiare e rilasciandolo quando termina di mangiare.
Quindi la procedura da richiamare quando il filosofo termina di mangiare è la seguente:

```
25 void posa(int i){
26  pthread_mutex_lock(&mutex);                    /* accesso in mutua esclusione */
27  stato[i]=PENSA;                                /* aggiorna il suo stato */
28  pthread_mutex_unlock(&mutex_f[i]);             /* libera la sua risorsa */
29  pthread_mutex_unlock(&mutex);                  /* rilascia la regione critica */
30 )
```

Quando invece vuole iniziare a mangiare è necessario richiamare la procedura che, per esempio, ha la seguente struttura:

```
42 void prendi(int i){
43  pthread_mutex_lock(&mutex);                    /*  accesso in mutua esclusione */
44  printf("\nFILOSOFO %d: ho fame e aspetto le forchette ... ",i);
45  stato[i]=ATTESA;
46  controlloPosate(i);                            /* controlla e attende le posate*/
47  printf("\nFILOSOFO %d: ...ora prendo le forchette e mangio",i);
48  pthread_mutex_lock(&mutex_f[i]);               /*        occupa la sua risorsa */
49  pthread_mutex_unlock(&mutex);                  /*  rilascia la regione critica */
50 )
```

dove i controlli sulle posate libere e le operazioni che ne conseguono vengono fatte nella funzione controlloPosate.

Prova adesso!

Realizza la procedura controllo posate che garantisce l'assenza di starvation e deadlock al sistema, utilizzando per il filosofo la seguente procedura

```
52 void *filosofo(void *x){
53 int i=(int)x;
54   while(1){
55     pensa(i);
56     prendi(i);
57     mangia(i);
58     posa(i);
59  )
60 )
```

E il seguente main:

```
61
62 main(){
63  int i;
64  pthread_t filo[N];                             /* definizione dei thread    */
65  pthread_mutex_init(&mutex,NULL);               /* inizializzazione semafori */
66  for(i=0; i<N; i++)
67   pthread_mutex_init(&mutex_f[i],NULL);
68  for(i=0; i<N; i++){
69   pthread_create(&filo[i], NULL,(void *)filosofo,(void *)i); /* crea i thread */
70   sleep(1);
71  }
72  pthread_exit(0);
73 }
```

Ora modifica il numero di filosofi e manda in esecuzione il programma: cosa puoi osservare?

Esercitazioni di Laboratorio 4

I SEMAFORI IN JAVA

Info

I codici seguenti sono reperibili nel file Java_sincronizzazione_rar scaricabile dalla cartella materiali nella sezione del sito www.hoepliscuola.it riservata al presente volume.

Modello ad ambiente globale: interazione tra thread

In Java i thread hanno come naturale meccanismo di comunicazione il modello ad ambiente globale: vediamo un primo esempio di come i thread possono interagire tra loro comunicando nell'area dati comune del processo che li ha generati (ambiente globale) mediante due modalità: definiamo un oggetto di tipo integer nell'area dati del processo padre e passiamo il reference di tale oggetto al costruttore dei thread; successivamente, sfruttiamo le regole di scooping accedendo direttamente dai thread alle singole variabili.

Per garantire l'unicità di variabili e oggetti che dovranno essere condivisi è necessario dichiararli con la clausola static, cioè come variabili di classe.

In questo primo esempio comunichiamo attraverso entrambe le modalità descritte, e cioè:
- direttamente nell'area dati indirizzando la variabile `InComune1.x`;
- passando ai thread un reference di un oggetto `Contatore` creato nel processo padre (`conta`).

```
public class InComune1{
  // a) ambiente globale: visibili da ogni thread
  protected static int x = 100;

  public static void main(String [] args){
    //b) oggetto comune passato al thread
    Contatore conta = new Contatore(0,1);
    Thread  thr1 = new UnThread1("ali",conta);
    Thread  thr2 = new UnThread1("baba",conta);
    thr1.start();
    thr2.start();
  }
}
```

```
class Contatore{
  int valore;
  int passo;
  Contatore (int valore,int passo)  {
    this.valore=valore;
    this.passo =passo;
    System.out.println("il contatore è nato e vale "+this.getValore());
  }
  void su ()  {
    valore+=passo;
  }
  int getValore()  {
    return valore;
  }
}
```

In questo codice, una semplice classe `Contatore`, di cui abbiamo creato un oggetto e da cui abbiamo passato a entrambi i thread il reference conta, è la seguente: ▶

Nel costruttore del contatore è stata inserita l'istruzione `println()` unicamente per visualizzare dove viene creata la variabile valore (ovvero nel thread `main()`).

Vediamo ora la classe `UnThread1`: nella dichiarazione delle variabili di classe è presente l'identificatore dell'oggetto della classe `Contatore` e il reference di tale oggetto viene passato come parametro mediante il costruttore della classe `UnThread1` (insieme al nome da assegnare all'oggetto stesso, cioè al thread stesso).
Il corpo del thread, cioè il metodo `run()`, è costituito da un ciclo infinito (che poi nell'esempio viene interrotto alla nona interazione!) che:

- dapprima visualizza il contenuto delle variabili locali e delle variabili globali;
- quindi invoca `conta.su()` (che è un metodo della classe `Contatore`) sull'oggetto comune: il risultato è l'incremento del valore della variabile valore (attributo di tale oggetto);

```
class UnThread1 extends Thread{
  // stato dell'oggetto
  private int inizia = 0;       // variabile interne di servizio inizializzate dal costruttore
  private int delta  = 1;
  private String mionome = "";
  Contatore conta;
  // costruttore
  UnThread1 (String nomethread, Contatore conta){
     this.mionome  = nomethread;
     this.conta    = conta;
  }
  // corpo del thread
  public void run()  {
    for(;;){                // ripeti per sempre
      System.out.print("Io sono "+mionome+" inizia vale "+inizia+" mentre x è "+InComune1.x);
      System.out.println("  => il contatore vale "+conta.getValore());
      inizia += delta;
      //modifica direttamente le variabili della classe che lo crea (sconsigliato)
      conta.su();
      //modifica direttamente le variabili della classe che lo crea (sconsigliato)
      InComune1.x++;
      try { Thread.sleep(500); }
      catch (InterruptedException e) { System.out.println(e); }
      if (inizia > 5)                    // termiuna dopo 5 ripetizioni
        return;
    }
  }
}
```

Da quest'ultimo codice, otteniamo il seguente output:

```
BlueJ: Terminal Window - UD-SINCRONIZZAZIONE
Options
il contatore è nato e vale 0
Io sono baba inizia vale 0 mentre x è 100  => il contatore vale 0
Io sono ali inizia vale 0 mentre x è 101  => il contatore vale 1
Io sono baba inizia vale 1 mentre x è 102  => il contatore vale 2
Io sono ali inizia vale 1 mentre x è 103  => il contatore vale 3
Io sono baba inizia vale 2 mentre x è 104Io sono ali inizia vale 2 mentre x è 104  => il contatore vale 4
  => il contatore vale 5
Io sono baba inizia vale 3 mentre x è 106  => il contatore vale 6
Io sono ali inizia vale 3 mentre x è 106  => il contatore vale 7
Io sono ali inizia vale 4 mentre x è 108  => il contatore vale 8
Io sono baba inizia vale 4 mentre x è 108  => il contatore vale 9
Io sono ali inizia vale 5 mentre x è 110  => il contatore vale 10
Io sono baba inizia vale 5 mentre x è 110  => il contatore vale 11
```

Osserviamo come le variabili interne di conteggio `inizia` sono "locali" ai singoli thread in quanto ogni thread incrementa la propria, mentre sia la variabile `valore` dell'oggetto `conta` sia la variabile `x` della classe `InComune1` a ogni iterazione di ogni thread "subiscono" un incremento, cioè sono *effettivamente comuni ai due thread*.

Ma l'output mostra come si sia verificato qualche problema di interleaving: se ripetiamo più volte l'esecuzione del programma probabilmente avremo sempre sequenze diverse.
È necessario introdurre i meccanismi che permettono la corretta gestione della risorsa condivisa.

Nei thread, la risorsa condivisa è costituita dalle variabili globali e quindi implementiamo le primitive per accedere e utilizzare i dati condivisi. Tutta la libreria di oggetti di Java è thread-safe e quindi tutte le classi che scriveremo genereranno oggetti che godono di tale proprietà, permettendoci l'accesso condiviso.

CLASSE THREAD-SAFE

Una classe è detta thread-safe se vi si può accedere contemporaneamente da un insieme di thread.

In Java i thread utilizzano i monitor di Hoare come meccanismo di sincronizzazione nel modello ad ambiente globale: cioè ogni istanza di una classe threadsafe (e quindi di ciascuna classe) ha associato un monitor e i metodi di tale classe sono eseguiti in mutua esclusione.

L'implementazione avviene modificando la notazione dei metodi nel modo seguente. ▶

```
public class Sincronizzata{
  ...
  public synchronized void metodoA() {
    //sezione critica
   }
  ...
  public synchronized int metodoB() {
    //sezione critica
  }
 ...
}
```

Il qualificatore `synchronized` garantisce che solo un thread alla volta possa eseguire tale metodo, mandando gli altri in uno stato di attesa: l'insieme dei thread in attesa per l'utilizzo di un oggetto prende il nome di wait-set.

Se il metodo è lungo, potrebbe di conseguenza risultare lunga l'attesa degli altri all'ingresso del monitor: è anche possibile restringere la regione critica a una porzione del metodo mediante il costrutto:

```
  synchronized(this) {
  //regione critica
  ...
  ...
}
```

Realizzazione dei semafori in Java

Java non ammette semafori espliciti come elementi del linguaggio, ma proprio per la sua natura di linguaggio a oggetti permette di costruire una classe che realizza i semafori di Djikstra.

- A tal fine la classe Object mette a disposizione due metodi, `wait()` e `notify()`, che ci permettono di scrivere le primitive P(S) e le V(S):
- l'operazione `wait()` sospende l'esecuzione del thread e lo colloca nella lista in attesa del verificarsi di una condizione; il thread rilascia la risorsa e rilascia la mutua esclusione;

- l'operazione `notify()` segnala che si è verificata una modifica in una condizione, e lo segnala alla lista dei thread sospesi su quella condizione, o meglio su quell'oggetto (wait-set), in modo tale che uno di essi possa riprendere l'esecuzione dallo stato di attesa.

È necessario poi porre particolare attenzione a quanto di seguito indicato:

- l'istruzione `wait()` deve essere inserita in un costrutto `try/catch` in quanto può essere sospesa dal metodo interrupt che genera un'eccezione `Interrupted-Exception`;
- sia `wait()` che `notify()` devono essere chiamate da un metodo `synchronized`, altrimenti si incorre nell'eccezione `IllegalMonitorStateException`.

Scriviamo ora la classe semaforo dove realizziamo le due primitive P() e V() con due metodi ad accesso in mutua esclusione:

```
class Semaforo {
  int valore;
  public Semaforo(int v){
    valore=v;
  }

  synchronized public void P(){
    while (valore==0){                    // semaforo rosso
      try { wait();}                      // il thread si sospende
      catch(InterruptedException e){}
    }
    valore--;                             // pone il semaforo a rosso
  } //end P

  synchronized public void V(){
    valore++;                             // pone semaforo a verde
    notify();                             // risveglia thread in coda
  } //end V
}
```

Il costruttore ci permette di inizializzare il semaforo a rosso oppure a verde.

ESEMPIO 19 ***Produttore e consumatore regolato da semafori***

Realizziamo una semplice situazione di produttore/consumatore, dove il produttore scrive in sequenza N numeri e il consumatore li visualizza sullo schermo.

Dichiariamo una variabile globale che servirà da buffer per i due thread e due semafori che ne permetteranno l'accesso, rispettivamente il primo che segnala quando il buffer è pieno e il secondo che segnala quando il buffer è vuoto, e li passiamo come parametri ai due thread:

```
public class ProdConsSem {
  protected static int buffer;          // variabile condivisa globale
  public static void main(String args[]){
    Semaforo pieno = new Semaforo(0);   //inizialmente rosso
    Semaforo vuoto = new Semaforo(1);   //inizialmente vuoto
    Produci pr=new Produci(pieno,vuoto);
    Consuma co=new Consuma(pieno,vuoto);
    pr.start();
    co.start();
  } //end main
}
```

Il thread Produci si limita a scrivere un dato nel buffer quando trova verde il semaforo che indica buffer vuoto: altrimenti si sospende e, quando riesce a scrivere, setta a verde il semaforo che risveglia il consumatore. ▶

```
class Produci extends Thread {
  Semaforo pieno;
  Semaforo vuoto;
  int tanti= 5;              // numero di elementi da scrivere

  public Produci(Semaforo s1,Semaforo s2){
    pieno=s1;
    vuoto=s2;
  }

  public void run() {
    int k;
    for (k=0;k<tanti;k++){
      vuoto.P();
      ProdConsSem.buffer=k;          // produce un numero
      System.out.println("Scrittore: dato scritto :"+k);
      pieno.V();
      try   (Thread.sleep(500);)
      catch (Exception e){ }
    }
  } //fine run
}
```

Il thread Consuma si limita ad aspettare che un dato sia pronto da leggere e quando lo ha consumato setta a verde il semaforo che risveglia il produttore. ▶

```
class Consuma extends Thread {
  Semaforo pieno;
  Semaforo vuoto;
  int dato;
  public Consuma(Semaforo s1,Semaforo s2){
     pieno=s1;
     vuoto=s2;
  }

  public void run() {
    while (true){
      pieno.P();
      dato=ProdConsSem.buffer;    // consuma un numero
      System.out.println("Lettore: dato letto "+dato);
      vuoto.V();
    }
  } //fine run
}
```

Mandando in esecuzione il programma, si ottiene il seguente output:

Prova adesso!

1. Modifica il programma utilizzando come buffer un oggetto, per esempio del tipo contatore come descritto nel primo esempio.

2. Modifica il programma precedente in modo che siano presenti due produttori e due consumatori: il primo produttore incrementa il dato di una sola unità mente il secondo lo raddoppia. A video i consumatori visualizzano oltre al valore letto anche il proprio nome.

3. Realizza un programma per sincronizzare un produttore che riempie una coda circolare di 10 elementi e tre consumatori che estraggono dalla coda un elemento ciascuno.
   Il produttore incrementa un contatore a ogni scrittura e il consumatore lo visualizza sullo schermo, assieme al proprio identificatore.

4. Successivamente realizza una situazione dove N produttori ed M consumatori utilizzano una risorsa con memoria limitata, per esempio composta da 10 elementi.
   Manda in esecuzione una possibile situazione di tre produttori e tre consumatori che rispettivamente inseriscono un numero progressivo e lo leggono dal buffer visualizzandolo sullo schermo.

5. Realizza un programma per sincronizzare tre processi che rispettivamente visualizzano il suono di una campana in modo da ottenere la sequenza infinita di "DIN" "DON" "DAN".

6. Realizza un programma che sincronizzi la situazione lettori/scrittori, dove più lettori possono contemporaneamente accedere alla risorsa condivisa mentre gli scrittori devono alternarsi, nei seguenti casi:
   - A privilegiando i lettori
   - B privilegiando gli scrittori
   - C alternando il più possibile lettori e scrittori

Esercitazioni di Laboratorio 5

I MONITOR IN JAVA

Realizzare i monitor

Per implementare i monitor in Java si realizza una classe con al suo interno i metodi di entry che accedono alle risorse condivise qualificati con synchronized: ▶

```
public class Monitor{
  <variabili condivise>
  ...
  public synchronized void entryA() {
    ... // regione critica
  }
  ...
  public synchronized void entryB() {
    ... // regione critica
  }
  ...

}
```

Quando si invocano le entry sincronizzate su di un oggetto Monitor, questo viene bloccato e quindi abbiamo mutua esclusione su di esso, dato che nessun altro processo può invocare nessun altro metodo dello stesso oggetto.
Le primitive `wait()` e `notify()` invocate all'interno di metodi `synchronized` evitano il verificarsi di race condition:

- `wait()` invocata senza parametri resta in attesa fino a che non viene risvegliata da una notify, in alternativa è possibile specificare un parametro che indica il numero di millisecondi da attendere prima del risveglio: `wait(long timeout)`
- `notify()` sblocca un processo ma non lo esegue subito, ossia permette al codice che ha chiamato la notify di continuare la sua esecuzione

> La primitiva `notify()` di Java sblocca uno dei processi in attesa, ma il chiamante non sa quale, dipende dalla JVM: esiste una ulteriore primitiva, la `notifyall()`, che sblocca tutti i processi in attesa sulla condizione; semplifica molto la programmazione, ma non è molto efficiente.

ESEMPIO 20 *Monitor che regola un buffer condiviso*

Realizziamo per esempio un monitor che regola l'accesso a una risorsa dato che verrà in seguito utilizzata negli esempi produttori/consumatori e lettori/scrittori.

Scriviamo due entry, una per prelevare il dato e l'altra per inserirlo, che aggiornano lo stato di una variabile booleana che segnala se il dato è disponibile o meno.

```
class Buffer {
  private int dato ;                          //variabile contenente il dato
  private boolean disponibile = false;        //indica se il dato è disponibile

  public synchronized int togli () {
    while (disponibile == false) {
      try { wait(); }
      catch (InterruptedException e) {}
    }
    disponibile = false;                      //svuota il buffer
    notify();                                 //avvisa un processo tra i sospesi
    return dato;
  }

  public synchronized void metti (int valore) {
    while (disponibile == true) {             //non è stato svuotato il buffer
      try { wait(); }
      catch (InterruptedException e) {}
    }
    dato = valore;
    disponibile = true;
    notify();                                 //avvisa un processo tra i sospesi
   }
}
```

Un esempio di programma che utilizza questo monitor è il seguente, dove vengono creati un thread produttore e uno consumatore ai quali viene passato come parametro un oggetto "monitor":

```
public class UnProd_UnCons  {
  public static void main(String args[]) {
    Buffer buffer =new Buffer();
    Produci prod1=new Produci(buffer,1);  // produttore di nome 1
    Consuma cons1=new Consuma(buffer,1);  // consumatoredi nome 1
    prod1.start();
    cons1.start();
  }
}
```

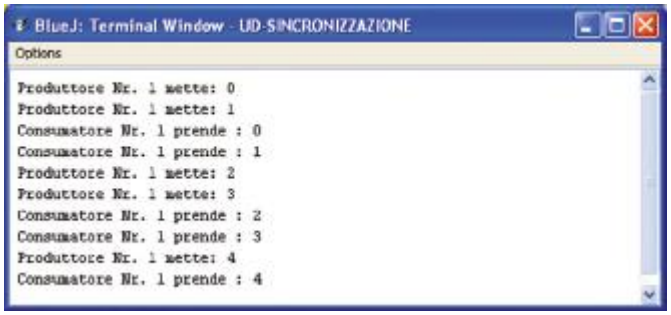
Le classi Produci e Consuma, di seguito riportate, hanno un costruttore che riceve come parametro anche un numero intero che viene utilizzato per dare un "nome" ai singoli thread: in questo caso, avendo un solo thread produttore e consumatore, a entrambi viene passato il valore 1 come parametro.

```
class Consuma extends Thread {
  private Buffer buffer;       // monitor per il dato condiviso
  private int numero;          // per dare un nome al consumatore
  public Consuma (Buffer buffer, int numero ) {
    this.numero=numero;
    this.buffer=buffer;
  }
  public void run() {
    for(int xx=0; xx<5; xx++) {
      int valore=buffer.togli();
      System.out.println("Consumatore Nr. " + numero + " prende : " + valore);
    }
  }
}
```

Nel metodo Run() di ciascuna classe viene effettuata la scrittura e la lettura per cinque volte visualizzando a schermo quanto scritto e quanto letto:

```
class Produci extends Thread {
  private Buffer buffer;       // monitor per il dato condiviso
  private int numero;          // per dare un nome al produttore
  public Produci ( Buffer buffer, int numero ) {
    this.numero=numero;
    this.buffer=buffer;
  }
  public void run() {
    for(int xx=0; xx<5; xx++) {
      buffer.metti(xx);
      System.out.println("Produttore Nr. " + numero + " mette: " + xx);
    }
  }
}//Produttore di cinque numeri interi
```

Mandando il programma in esecuzione si ottiene l'output riportato a fianco: ▶



L'output a schermo non visualizza "cronologicamente" gli eventi ma è fortemente dipendente dalla schedulazione: è comunque evidente che un dato per essere letto deve essere stato prodotto, a prescindere dall'ordine dell'echo che i diversi thread presentano a schermo.

Prova adesso!

1. Modifica l'esempio precedente per generalizzare il problema nel caso che siano presenti due produttori e tre consumatori.
   Quindi sostituisci il buffer con un nuovo monitor, bufferCircolare con dimensione BUFFER_SIZE, e utilizza due variabili (`int readpos, writepos;`) per leggere/scrivere nel buffer:
   - `writepos` è il puntatore alla prima posizione libera;
   - `readpos` è il puntatore al primo elemento occupato.

2. Modifica l'esempio precedente per generalizzare il problema a N produttori e N consumatori che utilizzano sempre un buffer circolare con dimensione BUFFER_SIZE.
   Effettua le necessarie modifiche al monitor e ai processi di prova nonché al main che deve generare rispettivamente:
   - 5 processi produttori e 10 processi consumatori;
   - 10 processi produttori e 5 processi consumatori;
   - 10 processi produttori e 10 processi consumatori.

   Discuti i risultati che hai ottenuto.

3 Realizza un sistema in cui un produttore generi i numeri di Fibonacci (il primo è 1, il secondo è 2, ogni numero successivo è la somma dei due precedenti) introducendo il risultato in un buffer; il consumatore legge il numero e:
- se il numero è corretto, prosegue regolarmente la lettura finché arriva al termine n-esimo;
- se il numero non è stato aggiornato… aspetta;
- al termine, visualizza tutti i numeri della sequenza.

4 Realizza un programma che sincronizzi la situazione lettori/scrittori, dove più lettori possono contemporaneamente accedere alla risorsa condivisa mentre gli scrittori devono alternarsi, nei seguenti casi:
- privilegiando i lettori;
- privilegiando gli scrittori;
- alternando il più possibile lettori e scrittori.

5 Realizza un'applicazione concorrente per la gestione dello studio di un veterinario, unico per cani e gatti, dove ogni utente del veterinario (cane o gatto) è rappresentato da un thread e il suo studio come una risorsa. La politica di sincronizzazione tra i processi dovrà garantire che:
- nello studio non vi siano contemporaneamente cani e gatti.
- nell'accesso allo studio, i gatti abbiamo la priorità sui cani.

Si supponga inoltre che lo studio abbia una capacità limitata a N animali. Il numero C di cani, quello dei gatti G e la capacità N dovranno essere passati alla riga di comando.

6 In un'autocarrozzeria si devono verniciare alcune auto: per ogni auto si deve alternare un certo numero di fasi di ceratura con fasi di lucidatura che vengono eseguite da due operai che sono specializzati, rispettivamente, nel processo di ceratura e in quello di lucidatura.
Si deve simulare il comportamento dall'autocarrozzeria in JAVA:
- si attivino due thread che simulino il comportamento dei due operai;
- si definisca l'oggetto condiviso Auto, tramite cui interagiscono i due thread, con i metodi per la corretta sincronizzazione tra i due thread.

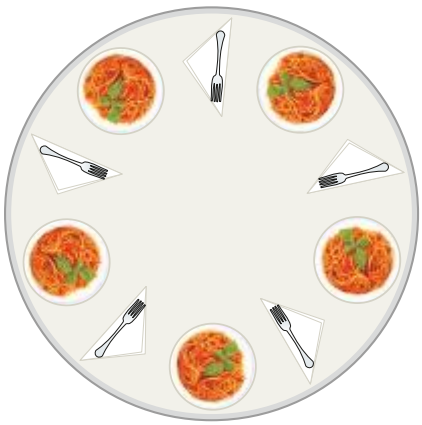
Il programma di prova Autocarrozzeria deve creare un oggetto Auto, attivare i due thread, passando a ciascuno di essi l'oggetto condiviso, lasciarli lavorare per un certo numero di secondi, quindi interromperli.

Esercitazioni di Laboratorio 6

IL DEADLOCK IN JAVA

Filosofi e deadlock

Si vuole realizzare in linguaggio Java la soluzione del problema dei 5 filosofi utilizzando un monitor Tavolo sul quale sono posti i piatti e le forchette.



Ciascun filosofo, numerato con un valore da 0 a N (possiamo poi provare il caso di N filosofi) può compiere tre attività: pensare, mangiare e, nel caso non trovasse disponibili le forchette, aspettare. ▶

La classe Filosofo è la seguente, dove viene passato al costruttore l'oggetto tavola e il nome che assegniamo al filosofo:

```
class Filosofo extends Thread{
  private Tavola tavola;      // monitor tavola con forchette
  private int nrFil;          // nome del filosofo

  public Filosofo(Tavola t, int nome){
    tavola=t;
    nrFil=nome;
  }

  public void run(){
    while(true) {
      // prima attivita' del filosofo: mangia
      tavola.prendiForchette(nrFil);            // entry del monitor
      System.out.println("il filosofo "+nrFil+" mangia");
      try {Thread.sleep(1000);}
      catch (InterruptedException e) {};

      // seconda attivita' del filosofo: pensa
      tavola.lasciaFochette(nrFil);             // entry del monitor
      System.out.println("il filosofo "+nrFil+" pensa");
      try {Thread.sleep(1000);}
      catch (InterruptedException e) {};

    }
  }
}
```

Nel metodo `run()` sono evidenziate le due attività del filosofo: quando vuole iniziare a mangiare accede al monitor con la entry `tavola.prendiForchette()` che verifica se il filosofo `nrFil` è in grado di prendere contemporaneamente entrambe le forchette, in modo da evitare la situazione di dealock dato che in questo modo è stata rimossa la condizione di hold & wait.

Quindi mangia per un determinato tempo e successivamente rilascia le forchette utilizzando una seconda entry del monitor, la `tavola.lasciaForchette()` e inizia a pensare, per poi ripetere all'infinito queste due operazioni.

La classe di prova crea i cinque filosofi, una istanza della classe monitor tavolo e avvia l'esecuzione dei singoli thread:

```
public class FilosofiCena{
  public static void main(String[] args){
    Filosofo filosofi[];
    filosofi=new Filosofo[5];
    Tavola tavola =new Tavola();
    for (int n=0;n<5;n++)                    // creazione dei 5 filosofi
      filosofi[n]=new Filosofo(tavola,n);

    for (int n=0;n<5;n++)                    // avvio dei 5 filosofi
      filosofi[n].start();

  }
}
```

Prova adesso!

Realizza il monitor Tavola dove sono presenti le forchette e le due procedure/entry che permettono ai filosofi di prendere e rilasciare le forchette, secondo il seguente schema:

```
class Tavola{
  int forchette[];                  // risorse condivise
  public Tavola() {
    forchette=new int[5];
    <inizializza le forchette>
  }

  public synchronized void prendiForchette(int nrFilosofo){
     <se le forchette sono occupate>
     <il filosofo si sospende>
     <altrimenti prende le forchette>
  }

  public synchronized void lasciaFochette(int nrFilosofo){
    <rilascia le forchette>
    notifyAll();                    // risveglia i filosofi in attesa
  }
}
```

Ora modifica il numero di filosofi e manda in esecuzione il programma: cosa puoi osservare?

Verifichiamo le competenze

1 Un ristorante Fast-Food somministra due sole portate: hamburger e/o patate fritte.

Ogni portata viene preparata da un cuoco che interagisce con un cameriere attraverso uno scaffale, dove possono essere accumulati fino a 10 piatti pronti per essere serviti. Il cuoco prepara le ordinazioni in sequenza e le depone sullo scaffale, eventualmente attendendo che ci sia un posto disponibile. Il generico cliente fa il suo ordine all'inserviente e quindi attende la consegna dell'hamburger, delle patate oppure di entrambe le portate.
Gli ordini vengono accettati dal cameriere che li inoltra al cuoco e, quando sono disponibili sullo scaffale, li consegna ai clienti.
Tenendo presente che la cottura di un hamburger è di 3 minuti mentre quella delle patate e di 5 minuti e che nel caso di ordinazioni multiple queste devono essere consegnate contemporaneamente al cliente, si realizzi un sistema in grado di sincronizzare i processi escludendo la possibilità che si verifichi una situazione di deadlock.

2 Nel ristorante Fast-Food dell'esercizio precedente, si aggiunga anche la possibilità di servire ai clienti una bevanda.

Il barista riceve l'ordinazione del cliente contemporaneamente al cuoco, prepara il cocktail in 4 minuti e lo posiziona sullo scaffale, occupando una delle NRMAX posizioni disponibili.
Si realizzi un sistema che possa gestire la sincronizzazione evitando sia problemi di starvation che di dealock.

3 Nel ristorante Fast-Food dell'esercizio precedente, si aggiunga anche la possibilità di servire i clienti per tavolo, cioè raccogliendo più ordinativi contemporaneamente per ciascuna portata disponibile: N hamburger, M patatine e X bevande.

Il barista deve iniziare la consegna ai diversi tavoli solo dopo che sono disponibili sullo scaffale tutti i componenti di ogni ordinazione, altrimenti aspetta e riceve nuove ordinazioni.
Si realizzi un sistema che possa gestire la sincronizzazione evitando sia problemi di starvation che di dealock.

3 LA SPECIFICA DEI REQUISITI SOFTWARE

UNITÀ DI APPRENDIMENTO

L 1 La specifica dei requisiti

L 2 La raccolta dei requisiti

L 3 Scenari e casi d'uso

L 4 La documentazione dei requisiti

OBIETTIVI

- Comprendere l'importanza della fase di analisi
- Avere il concetto di requisito utente e di sistema
- Avere il concetto di fase di esplorazione
- Conoscere le tecniche di esplorazione
- Avere il concetto di scenario e caso d'uso
- Analizzare il documento di Specifica dei Requisiti Software (SRS)
- Acquisire la struttura di un SRS
- Comprendere le caratteristiche SRS

ATTIVITÀ

- Individuare i requisiti utente
- Individuare i requisiti di sistema
- Utilizzare le tecniche di esplorazione
- Individuare gli scenari d'uso
- Saper descrivere in UML i casi d'uso
- Saper descrivere in UML il diagramma di contesto
- Saper documentare i casi d'uso
- Saper compilare il documento di Specifica dei Requisiti Software
- Validare le specifiche di un SRS

LEZIONE 1

LA SPECIFICA DEI REQUISITI

IN QUESTA LEZIONE IMPAREREMO...

- le diverse tipologie di requisiti software
- i requisiti utente e i requisiti di sistema
- i requisiti funzionali e i requisiti non funzionali

Premessa

Nel ciclo di vita del processo informatico le prime fasi che seguono lo studio di fattibilità riguardano la raccolta e l'analisi dei requisiti.

Con lo studio di fattibilità viene effettuata una valutazione preliminare di costi e benefici nella quale si stabilisce se è conveniente avviare il progetto oppure si possono individuare opzioni o alternative più adeguate, come utilizzare e adeguare altri prodotti presenti in commercio, e quindi valutare le risorse umane e finanziarie necessarie.
Questa fase generalmente si conclude con la presentazione di una offerta al cliente (preventivo o documento di fattibilità nel quale sono indicati i costi, i tempi ed eventualmente le modalità di sviluppo per ogni alternativa proposta).

La formulazione del preventivo impegna la software house alla sua realizzazione e al rispetto di quanto in esso descritto: generalmente questo viene compilato da un Software Architect Senior.

Nella successiva fase, quella di specifica dei requisiti, possiamo riconoscere le seguenti attività:

- analisi del problema: lo scopo dell'analisi del problema è quello di comprendere cosa deve fare il sistema software che si vuole realizzare; devono essere individuate le necessità degli utenti, la natura dell'ambiente operativo, le condizioni di mercato e/o di settore operativo, le condizioni di funzionamento;
- definizione delle funzionalità, dell'operatività, dei vincoli interni ed esterni alla azienda, delle prestazioni e di ogni caratteristica richiesta al sistema per soddisfare la necessità del cliente;

- redazione di un documento di Specifica dei Requisiti Software (SRS – *Software Requirement Specification*) che, in poche parole, non è altro che la trasformazione dei requisiti in un documento formalizzato che raccoglie le specifiche tecniche e funzionali che caratterizzano il sistema;
- convalida delle specifiche: prima di procedere alla realizzazione il SRS viene analizzato e rivisto con il committente per validare ogni singola specifica individuata.

La specifica dei requisiti risponde alla domanda *"che cosa deve fare il sistema?"* cioè stabilisce le funzionalità del sistema **senza** interessarsi di "**come**" queste funzioni verranno realizzate.

L'analisi e l'individuazione dei requisiti (◀ **Software requirements** ▶) è di fondamentale importanza per poter sviluppare un software di alta qualità: un errore in questa fase comporta necessariamente la presenza anche di molti errori nel sistema finale.

L'insieme delle attività che si "occupano" di requisiti prende anche il nome di ingegneria dei requisiti (requirements engineering).

In questa lezione inizieremo ad analizzare il concetto di requisito e ne studieremo alcune possibili classificazioni che ci permetteranno di comprenderne meglio le caratteristiche al fine di acquisire gli elementi essenziali per poter poi redigere correttamente l'SRS.

◀ **Software requirements** Def. IEEE Std 610.12 (1990):
(A) A condition or capability needed by a user to solve a problem or achieve an objective.
(B) A condition or capability that must be met or posessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
(C) A documented representation of a condition or capability as in definition (A) or (B). ▶

Requisiti software e stakeholder

Un *requisito* (dal latino *requisitus*, richiesto) è una proprietà richiesta, oppure auspicabile, del prodotto: il documento dei requisiti ha lo scopo di accogliere in forma organica una descrizione di tutte le proprietà desiderate e dalla sua formulazione dovrebbe essere chiaro se un requisito esprime una proprietà obbligatoria, oppure soltanto suggerita o auspicabile.

Possiamo dare una prima definizione di requisito:

REQUISITO
Ogni informazione (ottenuta in qualche modo) circa le funzionalità, i servizi, le modalità operative e di gestione del sistema da sviluppare.

ESEMPIO 1

Per esempio, per un sito web di commercio elettronico potremmo identificare, fra gli altri, i seguenti cinque requisiti:

- il sito *deve* permettere all'utente di inserire nel carrello d'acquisto i prodotti di cui sta valutando l'acquisto;
- il carrello *deve* poter contenere almeno 15 prodotti contemporaneamente;

- ogni scheda prodotto contenuta nel catalogo *deve* contenere una fotografia a colori del prodotto, il suo nome, il nome del produttore, il prezzo e una descrizione sintetica ma completa, 5 righe di testo al massimo;
- il pagamento *deve* essere effettuato sia con carte di credito elettroniche che tradizionali;
- l'intero processo di acquisto di un prodotto *dovrebbe* richiedere al massimo 5 minuti.

La definizione dei requisiti rappresenta l'analisi completa dei bisogni dell'utente e del dominio del problema allo scopo di definire quello che il sistema deve fare.
La raccolta dei requisiti è spesso considerata l'attività più difficile, perché richiede la collaborazione tra più gruppi di partecipanti con differenti background: questa fase deve coinvolgere (engagement) diverse persone sia del team di sviluppo (programmatori, sistemisti ecc.) che dell'azienda del cliente (committente, end-users, managers ecc.) e, a volte, può anche richiedere la consulenza di terze parti, come consulenti esterni, per esempio, per l'analisi della legislazione e/o di realizzazioni pre-esistenti.
Le diverse persone coinvolte in tale processo sono chiamate gli ◀ **stakeholder** ▶.

STAKEHOLDER

"Gli stakeholder – o portatori di interesse – sono tutte quelle persone o gruppi che influenzano e/o sono influenzati dalle attività di un'organizzazione, dai suoi prodotti o servizi e dai relativi risultati di performance". (Freeman, 1984)

◀ **Stakeholder** A person, group or organization that has interest or concern in an organization. Stakeholders can affect or be affected by the organization's actions, objectives and policies. ▶

Sono da considerarsi stakeholder tutte le persone in qualche modo interessate alla messa in opera del sistema, a ogni livello della organizzazione.

All'inizio del progetto gli stakeholder difficilmente hanno una chiara idea di quello che esattamente vogliono e usano un proprio linguaggio tecnico (tipico del dominio dell'attività dell'azienda) che generalmente non è noto all'analista che deve quindi per prima cosa studiare il loro lessico specifico per poter dialogare e comprendere correttamente quanto gli viene esposto.
D'altra parte il cliente e gli utenti finali sono esperti nel loro dominio ma hanno generalmente poca (o nulla) esperienza nello sviluppo del software e idee spesso confuse e vaghe sulle funzionalità che il nuovo sistema vuole realizzare.
Inoltre sia le funzioni svolte che i fabbisogni individuali sono diversi a ogni livello della azienda e spesso gli stakeholder effettuano richieste diverse che possono anche sfociare in requisiti in conflitto tra loro.

Ogni stakeholder può fornire una descrizione astratta e imprecisa del sistema che l'analista deve essere in grado di elaborare per ottenere una descrizione dettagliata e matematica dello stesso.

Nella analisi dei requisiti bisogna tener presente non solo le esigenze dell'azienda ma anche i fattori esterni al sistema stesso, come le esigenze sopraggiunte a causa di modifiche apportate nella organizzazione aziendale conseguenti allì'adeguamento a nuove politiche di mercato o a obblighi legislativi.
Anche per questo motivo i requisiti non sono "stabili", ma possono cambiare sia durante la fase di analisi che in tutto il ciclo di sviluppo della applicazione informatica.

La fase di analisi dei requisiti è particolarmente delicata e l'introduzione di errori in questa fase può portare anche al fallimento dell'intero progetto: in ogni caso questi errori sono difficili da individuare e spesso vengono scoperti "troppo tardi", nelle ultime fasi del processo di sviluppo del software, quando l'intervento per risolverli risulta essere molto oneroso.
I principali rischi sono quelli di "dimenticare" o ignorare una funzionalità, di implementare in modo errato o incompleto una richiesta, di realizzare interfacce utenti poco intuitive e difficili da usare.

Un suggerimento per identificare i requisiti rilevanti in un processo di progettazione di sistemi informatici viene dato dalla normativa ISO 13407, che riporta:
- le prestazioni richieste al nuovo sistema in relazione agli obiettivi operativi ed economico/finanziari;
- i requisiti normativi o legislativi rilevanti, compresi quelli relativi alla sicurezza e alla salute;
- la comunicazione e la cooperazione fra gli utenti e gli altri attori rilevanti;
- le attività degli utenti, inclusa la ripartizione dei compiti, il loro benessere e le loro motivazioni;
- le prestazioni dei diversi compiti;
- la progettazione dei flussi di lavoro e dell'organizzazione;
- la gestione del cambiamento indotto dal nuovo sistema, incluse le attività di addestramento e il personale coinvolto;
- la fattibilità delle diverse operazioni, comprese quelle di manutenzione;
- la progettazione dei posti di lavoro e la interfaccia uomo-computer.

Prima di affrontare lo studio di una metodologia standardizzata per la definizione dei requisiti è necessario introdurre una loro classificazione.

Zoom su...

STAKEHOLDER

Il termine stakeholder deriva da stake, bastoncino, e holder, possessore, custode. Viene da tempi in cui le scommesse venivano raccolte ricevendo da ogni scommettitore un bastoncino che lo rappresentava: il custode dei bastoncini era lo stakeholder. Successivamente il termine fu oggetto di uno slittamento di significato, e passò a denotare non più chi riceveva le scommesse, bensì chi le aveva effettuate: la persona coinvolta, il "soggetto interessato" alla scommessa. È questo secondo significato che è arrivato a noi, quando a partire dagli anni sessanta del novecento il concetto di stakeholder si è diffuso nelle teorie di management e di analisi organizzativa. Possiamo tradurre stakeholder con "parte in causa, parte interessata": interessata in quanto può ricevere effetti positivi o negativi da un progetto, da un nuovo prodotto o servizio, da un evento.

Classificazione dei requisiti

I requisiti software possono essere classificati secondo due diversi punti di vista:
- livello di dettaglio:
  - requisiti utente (user requirements);
  - requisiti di sistema (system requirements):
- tipo di requisito che rappresentano:
  - requisiti funzionali;
  - requisiti non funzionali;
  - requisiti di dominio.

Livello di dettaglio

Analizzando i requisiti sulla base del livello di dettaglio, per ciascuna delle due categorie individuate abbiamo livelli di astrazione e formalismo diversi:

- i requisiti utente sono quelli che "osserva il cliente", cioè le esigenze sentite dall'utente finale e descritte con il linguaggio del cliente: vengono sottoposti al team di sviluppo che ne propone una soluzione, a volte con diverse alternative (sono anche chiamati requisiti aperti);
- i requisiti di sistema sono quelli imposti da vincoli esistenti, come per esempio l'utilizzo di apparecchiature esistenti all'interno della azienda o di interfacciamento con sistemi aziendali già in funzione o anche di natura fiscale e/o legislativa (rispetto della normativa sulla sicurezza e sulla privacy, contabilità e bilancio secondo la normativa CEE ecc.).
  Sono solitamente molto strutturati e vincolanti, scritti in linguaggi tecnici e/o semi-formali e/o formali e non lasciano margini di "inventiva" data la loro natura (sono requisiti più restrittivi o requisiti chiusi): spesso non sono conosciuti all'utente ma noti solo al programmatore.

ESEMPIO 2

In un progetto dove vi è un unico requisito utente elenchiamo i possibili requisiti di sistema:

Requisito utente:

- l'applicazione deve permettere di rappresentare e visualizzare file esterni prodotti da altri pacchetti software.

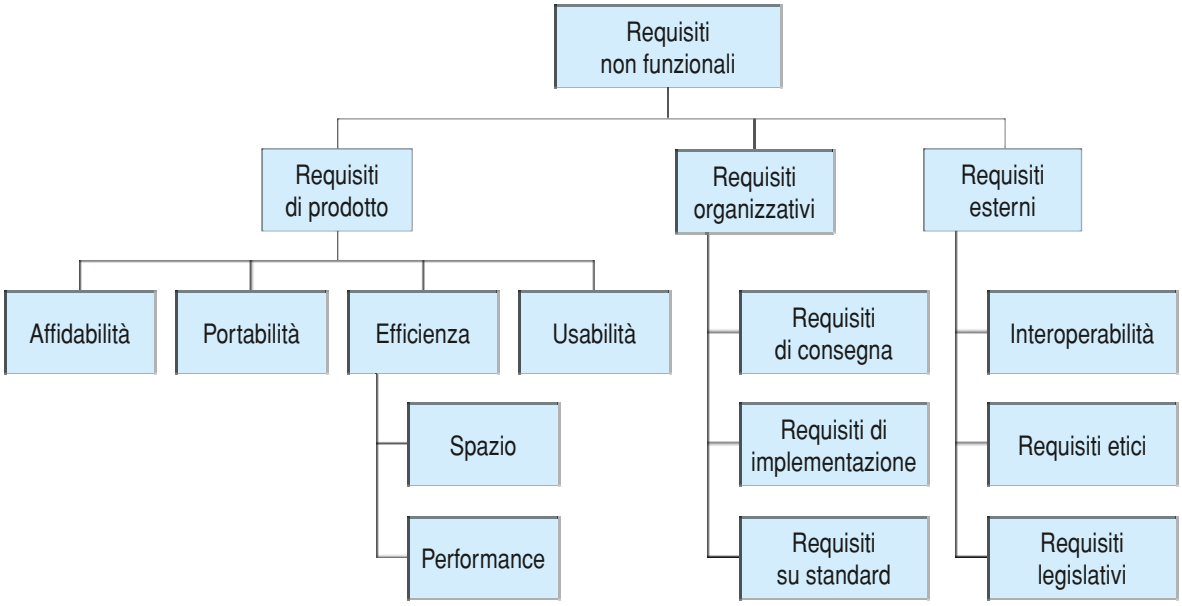
Requisito di sistema:

- l'utente deve poter definire il tipo dei file esterni;
- l'utente deve poter associare a ogni file esterno il prodotto che lo ha generato;
- a ogni tipo di file deve essere associata una specifica icona per visualizzarlo sullo schermo;
- l'icona che rappresenta il tipo di file esterno deve poter essere scelta dall'utente;
- selezionando l'icona, il sistema deve mandare in esecuzione una applicazione in grado di visualizzare il contenuto del file.

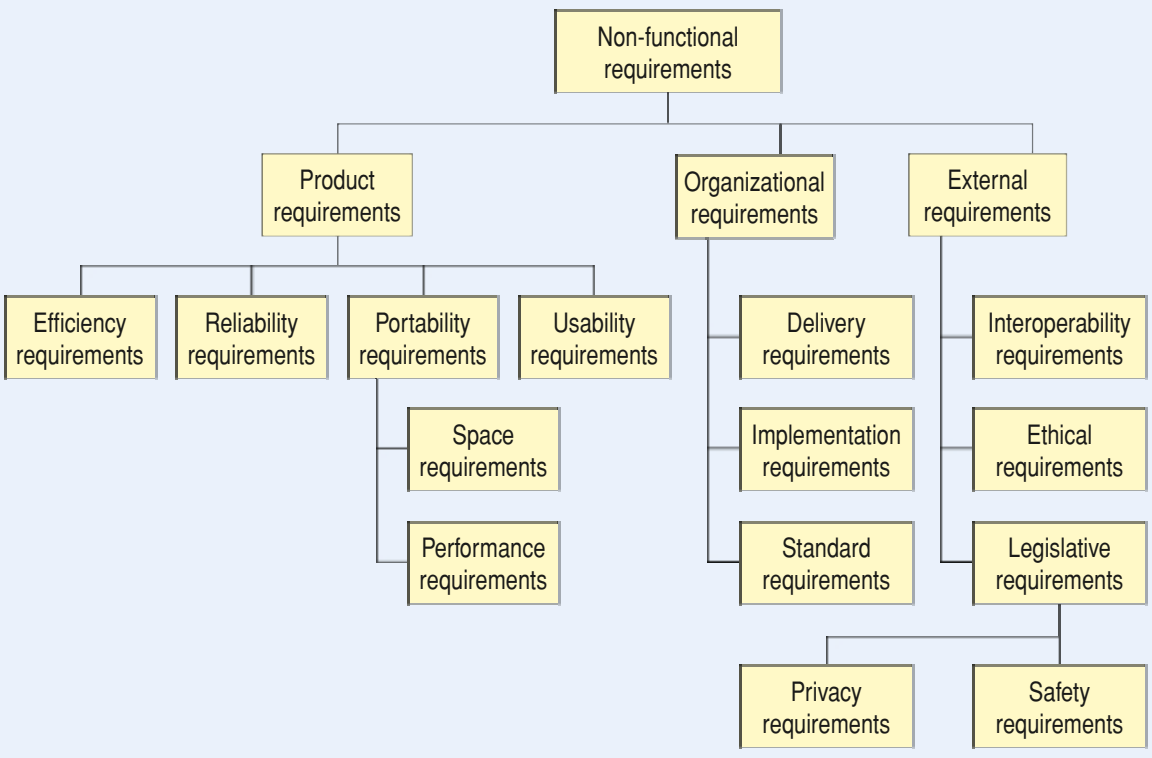
Tipo di requisito

Se invece analizziamo i requisiti sulla base del tipo, abbiamo tre possibili raggruppamenti:

- i requisiti funzionali (*Functional Requirement*) descrivono le funzionalità che il sistema deve avere e tutti i servizi che dovrà offrire agli utenti.
  I requisiti funzionali devono essere:
  – completi: devono indicare tutti i servizi richiesti dagli utenti;
  – coerenti: i requisiti non devono avere definizioni contraddittorie.
  Abbiamo già detto come per grossi sistemi sia difficile ottenere requisiti completi e coerenti dato che spesso i vari stakeholders hanno esigenze diverse, spesso in contrasto;
- i requisiti non funzionali (*Non Functional Requirement*) sono quelli imposti dalle modalità operative, dal ciclo di vita del prodotto", nel caso di un sistema di produzione, oppure dalla "catena del freddo" nel caso di un prodotto alimentare surgelato, o dalle precedenze nel triage in un pronto soccorso ecc., o del rispetto della normativa vigente: il sistema deve rispettare questi vincoli perché imposti dalla organizzazione e dall'esterno.
  Un esempio tipico è il tempo di risposta che deve avere il sistema per ciascuna operazione richiesta dall'utente oppure la possibilità di utilizzare il sistema sia su una piattaforma fissa (PC) che mobile (*smartphone*).
  Per i requisiti non-funzionali sono state proposte diverse classificazioni ◀ **Non-functional requirements classification** ▶: riportiamo di seguito quella di Sommerville dove definisce un primo livello composto da tre tipologie di requisiti: di *prodotto*, *organizzativi* ed *esterni*.



◀ **Non-functional requirements classification** Classification as an approach to requirements analysis; many people have proposed different requirements taxonomies: basically requirements can be classified into functional and non-functional. Each of these two categories can be subdivided into subclasses and so on: Sommerville gave a rather comprehensive taxonomy of non-functional requirements (NFR) for service-oriented software. ▶



- i requisiti di dominio (Domain Requirements) sono dipendenti dal dominio in cui il sistema deve operare, come la riservatezza nel caso di dati sensibili, le leggi della fisica e/o della tecnologia nel caso di impianti industriali, la sicurezza sul lavoro specifica per ogni tipo e settore di attività ecc. Un esempio tipico è la richiesta di login (user-id e password) per accedere a un'area dati protetta.

ESEMPIO 3

Analizziamo la seguente situazione per individuare le tipologie di requisiti.

Carta di credito
Una banca rilascia ai suoi clienti una carta di credito con la quale è possibile effettuare il pagamento degli acquisti e, presso uno sportello, effettuare di prelievo di contanti, visualizzare il saldo e l'estratto conto. Il sistema deve garantire un tempo di risposta inferiore al minuto, deve essere sviluppato su architettura X86 e deve essere disponibile a persone portatori di Handicap. Le operazioni di pagamento possono essere fatte entro un limite massimo mensile e quelle allo sportello richiedono una autenticazione tramite un codice segreto memorizzato sulla carta. Il sistema deve essere facilmente espandibile e adattabile alle future esigenze bancarie.

Già nella descrizione del progetto possiamo evidenziare la differenza tra requisiti funzionali, non funzionali e di dominio: i primi li indichiamo in verde, i secondi in rosso e gli ultimi in blu:

Una banca rilascia ai suoi clienti una carta di credito con la quale è possibile effettuare il pagamento degli acquisti e, presso uno sportello, effettuare di prelievo di contanti, visualizzare il saldo e l'estratto conto. Il sistema deve garantire un tempo di risposta inferiore al minuto, e deve essere sviluppato su architettura X86 e deve essere disponibile a persone portatori di Handicap. Le operazioni di pagamento possono essere fatte entro un limite massimo mensile e quelle allo sportello richiedono una autenticazione tramite un codice segreto memorizzato sulla carta. Il sistema deve essere facilmente espandibile e adattabile alle future esigenze bancarie.

Una seconda classificazione è chiamata modello FURPS (del 1987), ottenuto come acronimo di:
- Functionality: caratteristiche e capacità del programma, funzioni fornite, sicurezza dell'intero sistema;
- Usability (usabilità): legata alla semplicità di apprendimento e utilizzo del sistema da parte degli utenti (convenzioni per le interfacce utenti; organizzazione dell'Help in linea; tipologia e livello della manualistica);
- Reliability (affidabilità): il sistema deve garantire nel tempo le funzioni richieste per un determinato range di situazioni e fornire risposte sempre corrette (frequenza e severità delle failure, accuratezza degli output, capacità di recupero dalle failure, predicibilità del programma);
- Performance: sono i requisiti legati al tempo di risposta del sistema, cioè la velocità nel fornire i risultati, la quantità di risorse utilizzate, il throughput e quindi l'efficienza;
- Supportability (supportabilità): l'adattabilità del sistema alle modifiche che devono essere apportate durante il suo esercizio (manutenibilità, estendibilità, adattabilità, compatibilità e configurabilità).

A questi vanno aggiunti i vincoli (o pseudo-requisiti):
- implementazione: vincoli sull'implementazione del sistema, incluso l'uso di tool specifici, linguaggi di programmazione, o piattaforme hardware;
- interfacce: vincoli imposti da sistemi esterni, incluso sistemi legacy e formati di cambio;
- operazioni: vincoli sull'amministrazione e sulla gestione del sistema;
- packaging: vincoli sulla consegna del sistema (vincoli sui mezzi di installazione);
- legali: riguardano licenze, regolamentazioni e certificazioni.

I requisiti: l'anello debole dello sviluppo software

Un celebre studio effettuato dallo Standish Group su un campione di 8000 progetti riporta risultati sconfortanti: 16% di successi, 53% di fallimenti parziali (gravi problemi sulla funzionalità, sui costi e/o sui tempi), 31% di fallimenti completi (progetto cancellato).

Se si analizzano nel dettaglio le motivazioni (riportate nella seguente tabella) per le quali i progetti falliscono

Motivo del fallimento	Percentuale
Requisiti incompleti	13,1
Mancato coinvolgimento dell'utente	12,4
Mancanza di risorse	10,6
Attese irrealistiche	9,9
Mancanza del supporto della direzione	9,3
Cambiamento dei requisiti	8,7
Mancanza di pianificazione	8,1
Non serviva più	7,5

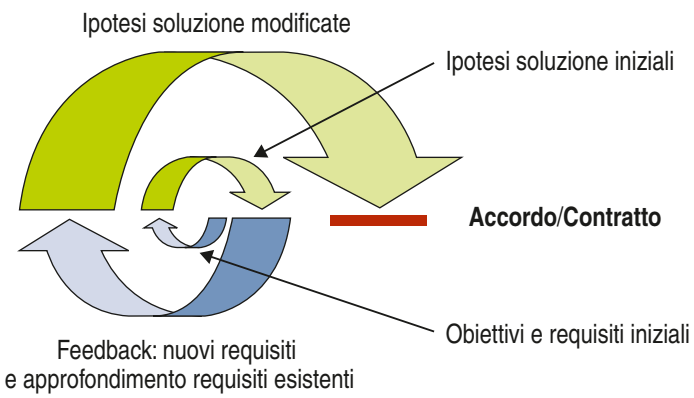
tra i primi otto fattori ne individuiamo cinque relativi a problemi connessi con i requisiti: requisiti incompleti, mancato coinvolgimento degli utenti, attese irrealistiche, cambiamento dei requisiti in corso d'opera, progetto cancellato perché non più utile.

L'intero ambito dei rapporti tra i progettisti di sistemi e i loro clienti risulta essere la causa prevalente del fallimento di un progetto software (51,6 % sull'80% delle principali cause).

> La maggior parte delle volte il fallimento del sistema viene "scoperto" solo alla messa in opera del sistema, individuando una non conformità rispetto ai requisiti attesi o concordati che ne determina la mancata accettazione del prodotto da parte del committente: solo in questo momento emergono i conflitti di interpretazione che incrinano i rapporti tra le parti.

Le principali cause che determinano questo ritardo nell'individuazione del problema sono:

- **A** insufficiente azione di interazione e discussione tra gli attori coinvolti;
- **B** insufficiente individuazione dei conflitti tra requisiti, sempre presenti in ogni progetto;
- **C** mancato coinvolgimento degli utenti nella verifica dei requisiti individuati dagli analisti;
- **D** mancato coinvolgimento degli utenti per esprimere feedback sulle ipotesi prospettate dai progettisti;



E insufficiente controllo sull'evoluzione dei requisiti durante l'intero ciclo di vita del sistema;
F rinvio della presa in carico del nuovo requisito sopraggiunto in corso d'opera a una release successiva;
G mancato accordo sui contenuti e/o i costi e/o i tempi durante la rinegoziazione per l'aggiunta di un requisito sopraggiunto in corso d'opera.

È importante introdurre una metodologia di specifica da utilizzarsi in tutte le fasi di analisi: la raccolta dei requisiti deve necessariamente essere fatta prima che i progettisti inizino a proporre qualche soluzione, durante la fase di realizzazione e dopo che il sistema è stato realizzato, durante la fase di collaudo.

Negli ultimi anni sono nati strumenti dedicati alla gestione dei requisiti (◀ **requirements management tools** ▶), che verranno trattati nella prossima lezione.

Il sistema finale deve essere raggiunto con una specifica soluzione che risponda in modo adeguato ai requisiti espressi dal committente e dagli altri attori coinvolti, deve avere caratteristiche comprensibili per tutti gli utenti, deve essere di semplice utilizzo e con funzionalità chiare, prive di ambiguità.

◀ **Requirements management tools** A **requirements management tool** is a software system that helps you manage the various manually intensive tasks in the requirements development and requirements management processes. Good requirements management tools (RM Tools) can help your team to save time and increase their productivity among other benefits. ▶

Verifica di requisiti

I requisiti funzionali sono abbastanza semplici da verificare: basta effettuare il collaudo del sistema con gli utilizzatori e verificare se tutte le loro aspettative sono state soddisfatte.

Anche i requisiti di dominio possono essere verificati agevolmente: si collaudano le interazioni del sistema con il "resto del mondo", cioè l'integrazione con il software aziendale preesistente, con terze parti e si verifica il rispetto delle normative di settore.

I requisiti non funzionali, invece, spesso indicati in modo generico dall'utente, possono risultare non quantificabili e difficili da verificare: a differenza dei precedenti, per i quali è possibile dire in "modo binario" se sono rispettati o meno, per questi generalmente è possibile dare un valore quantitativo del grado di soddisfacimento del requisito.
In tabella sono riportati, a titolo di esempio, degli indicatori quantificabili per alcune proprietà:

Proprietà	Misura
Velocità	Numero di transazioni per secondo Tempo di risposta di un evento Tempo di refresh dello schermo
Dimensione	Occupazione disco fisso programma Occupazione disco fisso per i dati Occupazione memoria RAM
Semplicità d'uso	Tempo richiesto per la formazione Dimensione della documentazione Numero di pagine di help on line

Proprietà	Misura
Affidabilità	Frequenza delle failure Gravità di una failure Accuratezza degli output Capacità di recupero dalle failure Predicibilità del programma
Robustezza	Tempo di riavvio dopo un guasto Percentuale di eventi che possono causare un crash del sistema Probabilità di danneggiamento dei dati
Portatilità	Percentuale di istruzioni dipendenti dai dispositivi HW Percentuale di istruzioni dipendenti dal SO

Prova adesso!

Data la seguente situazione, si richiede di individuare ogni tipo di requisito e, per i requisiti non funzionali, di predisporre una tabella per la valutazione quantitativa:

Si deve realizzare un sistema per archiviare i dati di una biblioteca, in particolare quelli relativi ai libri, ai giornali, alle riviste, ai video, ai nastri audio e ai CD-ROM.
Il sistema dovrà permettere agli utenti di fare delle ricerche per titolo, autore, genere o codice ISBN grazie a una interfaccia realizzata per un browser Internet.
L'accesso al sistema deve avvenire mediante autenticazione con smart-card e ogni prestito può durare al massimo 30 giorni per i libri, 7 giorni per i rimanenti articoli disponibili: un utente può avere contemporaneamente in consegna fino a 10 articoli.
Il sistema dovrà gestire almeno 20 transazioni per secondo e deve essere consultabile anche mediante smartphone.

Verifichiamo le conoscenze

Esercizi a scelta multipla

1 **Nella fase di specifica dei requisiti possiamo riconoscere le seguenti attività (indica quella errata):**

a) analisi del problema
b) definizione delle funzionalità
c) definizione del preventivo
d) redazione di un documento SRS
e) convalida delle specifiche

2 **Quale tra le seguenti figure non si può considerare uno stakeholder?**

a) committente
b) end-users
c) managers
d) analista
e) programmatori
f) sistemisti
g) nessuno di quelli elencati

3 **Secondo il livello di dettaglio i requisiti software possono essere classificati in:**

a) requisiti funzionali
b) requisiti utente
c) requisiti di sistema
d) requisiti non funzionali
e) requisiti di dominio

4 **Secondo il tipo di requisito che rappresentano, i requisiti software possono essere classificati in:**

a) requisiti funzionali
b) requisiti utente
c) requisiti di sistema
d) requisiti non funzionali
e) requisiti di dominio

5 **Quali tra i seguenti non appartiene al primo livello della classificazione di Sommerville?**

a) di prodotto
b) di sistema
c) esterni
d) organizzativi

6 **Una classificazione è chiamata modello FURPS, ottenuto come acronimo di (indica quello errato):**

a) Usability
b) Functionality
c) Reliability
d) Portability
e) Supportability

Test vero/falso

1	La specifica dei requisiti risponde alla domanda "che cosa deve fare il sistema?".	V F
2	La validazione dei requisiti risponde alla domanda "come lo deve risolvere il sistema?".	V F
3	La definizione dei requisiti rappresenta anche l'analisi del dominio del sistema.	V F
4	Le diverse persone coinvolte in tale processo sono chiamate gli stakeholder.	V F
5	I requisiti devono essere stabili per tutta la durata dello sviluppo.	V F
6	Per i requisiti non-funzionali ricordiamo la classificazione di Sommerville.	V F
7	Un esempio di requisiti di sistema è la richiesta di login (user-id e password).	V F
8	La verifica dei requisiti non funzionali può essere espressa mediante un valore quantitativo.	V F

Verifichiamo le competenze

Date le seguenti situazioni, si richiede di individuare ogni tipo di requisito e per i requisiti non funzionali di predisporre una tabella per la valutazione quantitativa:

1 Si deve realizzare un software per la gestione di un telefono cellulare, in particolare per poter inserire un testo in due circostanze:
1) composizione di SMS;
2) immissione nuovo nome/numero in rubrica.

Esistono delle regole generali che disciplinano l'immissione del testo, per esempio:
- come si cambia fra maiuscole e minuscole;
- come si attiva/disattiva la modalità T9.

Gli SMS devono essere raggruppati per destinatario (sia quelli spediti che quelli ricevuti) e ricercati anche in base al giorno di trasmissione/ricezione.
La rubrica deve permettere l'immissione di più numeri per lo stesso nominativo, indicando per ciascuno di essi il gestore telefonico e il numero di minuti di conversazione effettuata.
L'accesso alla rubrica deve essere protetto da password.

2 Si deve realizzare un sistema per la gestione di un ufficio postale: descrivilo in termini di un insieme composto da due moduli funzionali (Posta e Banca) e per ciascuno di essi indica:
- i servizi che offre agli utenti;
- la modalità con cui gli utenti possono usufruirne.

Dopo aver individuato le possibili interazioni tra i due moduli, considera la possibilità di utilizzo di una parte di tali servizi tramite web: individua quali funzionalità possono essere comuni e quali limitazioni necessariamente devono essere definite.

3 Si deve realizzare un sistema per un ristorante per gestire i clienti, i tavoli (con il relativo numero di posti), le prenotazioni (effettuate dai clienti per un certo giorno e ora, e per un certo numero di persone). Alle prenotazioni vengono assegnati uno o più tavoli (divisi in fumatori/non fumatori), i camerieri (che servono i clienti al tavolo) e il conto (composto dalle singole portate ordinate).
Dei clienti interessano il nome e il numero di telefono, mentre dei camerieri interessa, nome e anni di servizio.
Ogni portata ha un suo costo unitario previsto dal listino e al cliente viene presentato il conto dove vengono indicate le singole portate col loro nome, il prezzo unitario e la quantità ordinata che permette di calcolare il totale per portata e il conto totale da pagare.

4 Si deve realizzare un sistema software per una azienda costituita da diversi dipartimenti dove sono addetti un certo numero di impiegati con mansioni differenti: ogni impiegato è in forza solo a uno specifico dipartimento, e di lui sono memorizzati i dati anagrafici e gli stipendi percepiti.
Ogni dipartimento ha un suo nome, un direttore, un numero di telefono, la data di afferenza di ognuno degli impiegati che vi lavorano per poter calcolare i costi mensili, in termini di risorse umane.
Gli impiegati partecipano inoltre a vari progetti aziendali, dei quali viene indicato il nome e il budget: in questo caso lo stipendio non è a carico del dipartimento ma del progetto.

5 Si vuole progettare un sistema per la gestione di un campionato di calcio. Il sistema deve consentire la creazione del calendario delle partite e la designazione degli arbitri, con codice fiscale, nome, cognome e città di nascita. Il calendario è composto da un certo numero X di gironi, ognuno dei quali è composto da Y giornate, a loro volta composte da partite tra K squadre. Delle partite interessano il numero progressivo in schedina, la data, l'arbitro designato, le squadre (casa e ospite) e il risultato finale (che l'arbitro avrà il compito di memorizzare nel sistema, a fine gara, tramite un SMS).
Ogni squadra ha un proprio dirigente e un allenatore che decide quali giocatori convocare per le varie partite: le squadre sono individuate da un nome, dalla città e regione di provenienza.
Il sistema deve anche gestire la classifica generale e dei marcatori, consultabile da Internet sia con un PC o mediante uno smartphone.

LEZIONE 2

LA RACCOLTA DEI REQUISITI

IN QUESTA LEZIONE IMPAREREMO...

- la fase di esplorazione
- le tecniche di esplorazione

Premessa

Per poter effettuare la stesura del documento SRS di Specifica dei Requisiti Software è necessario effettuare la raccolta dei requisiti (◀ **Products requirements** ▶) e la loro analisi.

Abbiamo visto che queste due fasi rientrano nella ingegneria dei requisiti (requirements engineering), che possiamo ricordare come una sequenza di quattro attività:

- raccolta dei requisiti;
- analisi dei requisiti;
- stesura della documentazione dei requisiti SRS;
- verifica e approvazione dei requisiti.

La raccolta dei requisiti è considerata l'attività più difficile, perché richiede la collaborazione tra più gruppi di partecipanti con differenti background.

◀ **Product requirements** "The most difficult part of building a software system is to decide, precisely, what must be built. No other part of the work can undermine so badly the resulting software if not done correctly. No other part is so difficult to fix later." (Fred Brook) ▶

◀ Fred Brook Frederick Phillips Brooks Jr., (nato il 19 aprile 1931) è un ingegnere informatico, conosciuto per lo sviluppo del sistema operativo OS/360 della famiglia di computer IBM System/360: ne descrive il processo nel libro *The Mythical Man-Month*. Brooks attualmente si occupa di ingegneria del software e ha ricevuto numerosi riconoscimenti, tra cui la *National Medal of Technology* nel 1985 e il **Premio Turing** nel 1999. ▶

In questa lezione ci occuperemo dei primi due punti, cioè delle raccolta e dall'analisi dei requisiti, che prende anche il nome di fase di esplorazione.

Tipi di raccolta dei requisiti

A seconda del tipo di progetto la raccolta di requisiti degli attori potrebbe essere integrata da altre analisi; sostanzialmente abbiamo tre tipi di realizzazione:

- greenfield engineering: questa è la situazione in cui lo sviluppo parte da zero e nella azienda non esiste un precedente sistema da sostituire o integrare: la "fonte" dei requisiti è l'azienda committente, nella figura del cliente e degli *stakeholder* (attori); lo sviluppatore, però, deve anche valutare e ricercare la disponibilità sul mercato di un prodotto in grado di soddisfare le esigenze individuate da confrontare e/o proporre in alternativa allo sviluppo ex-novo;
- re-engineering: in questa situazione esiste già nella azienda un sistema che deve essere riprogettato perché tecnologicamente obsoleto rispetto alle nuove tecnologie o per estendere le funzionalità del sistema: l'analisi del sistema esistente è alla base del nuovo progetto perché si possono analizzare i suoi pregi e difetti, possono essere evidenziate le funzionalità che devono "migrare" nel nuovo sistema e possono essere valutate, per migliorare quelle che non soddisfacevano gli utenti, e integrate con le nuove esigenze e con le nuove possibilità offerte dalla tecnologia;
- interface engineering: è questa la situazione dove si è nella impossibilità di sostituire completamente il software esistente ma si deve adeguare almeno l'interfaccia con l'utente con i nuovi ambienti operativi: le funzionalità non sono "toccate" e quindi non è necessario raccogliere nuovi requisiti perché rimane inalterato nello "strato inferiore" il ◀ sistema legacy ▶ con i suoi servizi che vengono interfacciati con il nuovo ambiente, riprogettando semplicemente le interfacce.

◀ Sistema legacy Il termine inglese *legacy* è utilizzato per indicare qualcosa di valore ottenuto attraverso un'eredità. Associato a un sistema informativo, pertanto, il termine *legacy* ne evidenzia le caratteristiche di "valore aziendale" e di "provenienza dal passato". Nell'ambito dei sistemi informativi il termine *legacy* non sta a indicare qualcosa di vecchio ma, al contrario, uno strumento su cui l'azienda fa affidamento anche per il futuro: la convivenza fra questi sistemi e quelli realizzati con nuovi paradigmi e tecnologie è un dato imprescindibile. ▶

Nella documentazione gli attori vengono rappresentati graficamente con i seguenti simboli

ESEMPIO 4

Situazione

Il Comune di XYZ intende automatizzare la gestione delle informazioni relative alle contravvenzioni elevate sul suo territorio. In particolare, intende dotare ogni vigile di un dispositivo palmare che gli consenta di comunicare al sistema informatico il veicolo a cui è stata comminata la contravvenzione, il luogo in cui è stata elevata e la natura dell'infrazione. Il sistema informatico provvederà a notificare, tramite posta ordinaria, la contravvenzione al cittadino interessato.
Il Comune bandisce una gara per la realizzazione e manutenzione del sistema, che viene vinta dalla Ditta ABC.

Quali sono gli attori coinvolti in questa applicazione software?

- Committente: Comune di XYZ
- Esperto del domino: funzionario del Comune, o altro professionista designato, esperto del Codice della Strada
- Utenti finali: vigili

- // personale della ditta ABC
- Progettista
- Analista
- Programmatore
- Manutentore

La fase di esplorazione

La raccolta dei requisiti è anche detta fase di esplorazione in quanto il problema da risolvere deve essere proprio esplorato, cioè "sviscerato" in ogni sua minima componente, analizzato e affrontato in ogni suo minimo aspetto per individuare tutti i bisogni e definirne le priorità.

> La letteratura anglosassone attribuisce a questa fase i termini *elicitation* o *discovery* cioè *elicitazione e scoperta poiché* spesso i requisiti vengono "scoperti" in quanto sono difficili da essere individuati.

Per poter produrre il documento dei requisiti è necessario raccogliere il maggior numero possibile d'informazioni sugli obiettivi e sulle necessità riguardo al sistema da costruire, e quindi procedere con l'esplorazione del sistema.

- Il primo passo della fase di esplorazione è quello di esaminare le richieste del committente, cioè colui che ha richiesto lo sviluppo del progetto, che ha approvato e sottoscritto il preventivo ed è di fatto il principale referente (cliente).
- Il secondo passo della fase di esplorazione consiste nella definizione degli *stakeholder* (attori), cioè di tutti coloro che, in un modo o nell'altro, hanno qualche interesse nel prodotto, o la cui attività sarà influenzata, direttamente o indirettamente, da esso: si procede con il loro coinvolgimento (engagement) al progetto.

Lo stakeholder Engagement

Engagement è un sostantivo che significa coinvolgimento, ma allo stesso tempo richiama al concetto del "dedicarsi, occuparsi", nel senso che gli interlocutori coinvolti nello sviluppo partecipano attivamente al processo.
Gli ingegneri del software devono avviare un processo di dialogo con gli attori che inneschi una comunicazione interattiva in modo da confrontare reciprocamente le diverse aspettative di ciascuno di loro per impostare o rivedere politiche e strategie dell'impresa, integrando eventualmente quello che emerge da questa attività: spesso il punto di vista dei dipendenti è diverso da quello della azienda e frequentemente solo chi operativamente esegue determinate attività è a conoscenza di particolari criticità.

> È importante dare a questa fase il giusto peso, senza sottovalutarne l'importanza: **coinvolgere** vuol dire **comunicare** interattivamente, confrontarsi, dialogare, saper ascoltare, prendere degli impegni verso gli attori e **non significa** fare dei sondaggi, proporre soluzioni, comunicare delle scelte fatte da altri.
> Gli attori devono sentirsi "importanti" per la realizzazione del progetto (come, di fatto, lo sono) e devono far parte a tutti gli effetti del gruppo di lavoro.

È però anche importante individuare criteri di selezione che garantiscano la rappresentatività e inclusività degli stakeholder affinché il processo porti a risultati utili alla definizione dei requisiti e quindi al successivo sviluppo del progetto.

È necessario conoscere tutti i "punti di vista" (viewpoint), cioè analizzare il sistema dalle prospettive di tutti i possibili suoi utilizzatori e/o componenti che possono interagire con esso per individuarne tutti i requisiti e comprendere come questi possono essere realizzati.

La raccolta dei requisiti dagli stakeholder viene anche chiamata requirements elicitation.

Tecniche di esplorazione

La tabella seguente riporta le tecniche principali che possono essere utilizzate nella fase di esplorazione per la raccolta dei requisiti.

Strumenti	Obiettivi	Vantaggi	Svantaggi
Interviste individuali	Esplorare determinati aspetti del problema e determinati punti di vista.	L'intervistatore può controllare il corso dell'intervista, orientandola verso quei temi sui quali l'intervistato è in grado di fornire i contributi più utili.	Richiedono molto tempo. Gli intervistati potrebbero evitare di esprimersi con franchezza su alcuni aspetti delicati.
Focus group	Mettere a fuoco un determinato argomento, sul quale possono esserci diversi punti di vista.	Fanno emergere le aree di consenso e di conflitto. Possono far emergere soluzioni condivise dal gruppo.	La loro conduzione richiede esperienza. Possono emergere figure dominanti che monopolizzano la discussione.
Osservazioni sul campo	Comprendere il contesto delle attività dell'utente.	Permettono di ottenere una consapevolezza sull'uso reale del prodotto che le altre tecniche non danno.	Possono essere difficili da effettuare e richiedere molto tempo e risorse.
Suggerimenti spontanei degli utenti	Individuare specifiche necessità di miglioramento di un prodotto.	Hanno bassi costi di raccolta. Possono essere molto specifici.	Hanno normalmente carattere episodico.
Questionari	Rispondere a domande specifiche.	Si possono raggiungere molte persone con poco sforzo.	Vanno progettati con grande accuratezza, in caso contrario le risposte potrebbero risultare poco informative. Il tasso di risposta può essere basso.
Analisi della concorrenza e delle best practices	Individuare le soluzioni migliori adottate nel settore di interesse.	Evitare di "reinventare la ruota" e ottenere vantaggio competitivo.	L'analisi di solito è costosa (tempo e risorse).
Scenari e casi d'uso	Descrivere ogni singola operazione che il sistema deve effettuare.	Sono poi utilizzati in fase di collaudo per verificare le funzionalità del sistema.	Gli intervistati spesso non sono in grado di descrivere le criticità.

Interviste individuali

La migliore fonte per reperire i requisiti è quella delle interviste individuali.
L'indagine inizia con l'intervista al cliente/committente che è il maggior referente del progetto (anche perché "è lui che paga") e quindi ha chiari gli obiettivi che devono essere perseguiti e i tempi di realizzazione del sistema.
Sarà inoltre lui stesso a indicare al team di analisi gli stakeholder più significativi per ogni *categoria di personale*, con i quali procedere con le interviste.

Sarà sempre lui che, alla fine della stesura del documento, verrà coinvolto nella revisione e validazione.
Più persone vengono intervistate e più possibilità ci sono di avere nuove indicazioni utili ma anche di ottenere suggerimenti contraddittori, dato che ciascuno osserva un requisito solo dal proprio punto di vista e lo esprime in termini diversi: è essenziale valutare caso per caso le persone intervistare e selezionarle in modo opportuno, dopo averle suddivise in gruppi omogenei di utenti.

ESEMPIO 5 *Modalità di individuazione degli attori*

L'individuazione degli attori può essere fatta seguendo il seguente campione di domande:
- chi o cosa usa il sistema? Che compito svolge?
- chi ottiene informazioni dal sistema e a chi ne fornisce?
- quali gruppi di utenti eseguono le principali funzioni del sistema?
- quali gruppi di utenti eseguono le funzioni secondarie, come la manutenzione e l'amministrazione?
- sono presenti funzioni o azioni che vengono eseguite a intervalli prestabiliti?
- sono presenti funzioni svolte da una sola persona?
- quali sono i dipendenti che da più anni sono nella organizzazione?
- chi sono gli attori esterni all'organizzazione che interagiscono con essa?
- con quale sistema hardware o software il sistema interagisce?

Bisogna anche tenere conto del tempo a disposizione per questa fase, quindi è necessario saper individuare quali sono le persone che sicuramente hanno informazioni importanti e quali, invece, possono essere consultate solo in fase di verifica del sistema utilizzando un semplice questionario.

È possibile effettuare interviste con diversi livelli di strutturazione:

A *interviste non strutturate*: sono dei dialoghi dove l'intervistatore pone un insieme di domande aperte prestabilite allo stakeholder e dalle risposte può prendere spunti per avviare una indagine approfondita su eventuali nuovi requisiti o aspetti diversi di operazioni già individuate.

ESEMPIO 6 *Domande da effettuare agli stakeholder*

Il colloquio può iniziare semplicemente con una sequenza di domande del tipo:
- quali sono le attività che svolgi regolarmente?
- descrivi il flusso normale delle tue attività.
- descrivi cosa può andar male nell'esecuzione del flusso normale.
- cosa non funziona nel sistema attuale?
- come intendi utilizzare il sistema?
- come credi che il sistema debba essere utilizzato?
- cosa ti aspetti quando il sistema parte?
- …

B *interviste strutturate*: sono simili ai questionari (che tratteremo in seguito) ma sono somministrati verbalmente dall'intervistatore e consistono in un insieme di domande specifiche su di una particolare funzione o requisito che è già stato individuato ma richiede una indagine su tutti i possibili utilizzatori.
Proprio per la loro natura strutturata sono utilizzate per ricavarne dati statistici facilmente interpretabili.

- *interviste semi-strutturate*: sono una via di mezzo delle prime due, cioè sono interviste che contengono sia risposte aperte che domande specifiche a risposta chiusa.

Non è semplice condurre una buona intervista: un fattore importante è l'esperienza dell'intervistatore che deve avere ottime capacità di relazione e deve essere in grado di mettere l'intervistato a proprio agio, garantirne l'anonimato delle risposte, parlare con lo stesso linguaggio dell'intervistato, senza suggerire le soluzioni ma cercando di individuare i requisiti.
La maggior difficoltà è quella di individuare i requisiti impliciti che le persone considerano ovvi, e quindi non evidenziano nel colloquio poiché spesso rivelano come le operazioni dovrebbero essere svolte "in teoria" e non come sono svolte "in pratica".

Questionari

È il metodo più semplice per ottenere con bassi costi delle informazioni in forma strutturata facilmente elaborabili in modo automatico per ottenere delle statistiche.
Sono composti da un insieme di domande alle quali gli utenti rispondono scegliendo una risposta tra le cinque proposte: *completamente d'accordo*, *d'accordo*, *incerto*, *in disaccordo*, *in completo disaccordo* (scala di Likert).
A ciascuna risposta è associato un numero compreso fra 1 e 5 e con questi valori si potrà calcolare la media delle risposte a ciascun gruppo di affermazioni correlate a uno stesso argomento.
Possono essere utilizzati solo in fase di consuntivazione per ottenere un giudizio sul sistema appena realizzato, prima della sua messa in funzione, anche perché sono facilmente realizzabili e somministrabili a tutti gli utenti (esistono software che realizzano questionari da sottoporre online producendo direttamente i risultati).
L'attendibilità delle risposte, però, generalmente è bassa.

Focus group

I *focus group* sono interviste di gruppo che vengono gestite sul modello dei ◀ **brainstorming** ▶ partendo con l'analisi di un singolo argomento e cercando di far esprimere tutti i partecipanti in modo da ricercare gli elementi di condivisione e di contrasto.
Il mediatore deve essere in grado di gestire il gruppo, soprattutto nei casi di divergenza di opinioni, facendo in modo che la discussione non degeneri in lite e annotando ogni osservazione utile per la definizione e/o puntualizzazione dei requisiti.

◀ **Brainstorming** Brainstorming is a group or individual creativity technique by which efforts are made to find a conclusion for a specific problem by gathering a list of ideas spontaneously contributed by its member(s). ▶

Osservazioni sul campo

L'osservazione sul campo consiste nell'affiancare l'utente nelle sue operazioni quotidiane per visionare direttamente quali operazioni svolge nella attività quotidiana: spesso l'attore fatica a descrivere il proprio lavoro, soprattutto se questo utilizza strumenti software dei quali non ha particolari conoscenze tecniche ma solo operative.
Spesso l'utente non è in grado di evidenziare le criticità delle sue funzioni e quindi l'analisi congiunta "sul campo" con il responsabile dello sviluppo permette di scoprire dettagli importanti per la definizione dei requisiti.
È una attività molto onerosa, che richiede molto tempo e una conoscenza preventiva delle singole attività per individuare quando è più significativo effettuare l'osservazione; inoltre vengono osservate le funzioni "già presenti" e quindi difficilmente permette di individuare nuove funzioni che potrebbero migliorare il lavoro.

Suggerimenti spontanei degli utenti

Gli utenti suggeriscono come il prodotto deve essere realizzato anche nella fase delle interviste, ma inoltre possono ulteriormente contribuire fornendo indicazioni su appositi forum che vengono aperti sul web contestualmente all'inizio della attività di sviluppo: gli utenti segnalano spontaneamente miglioramenti desiderabili e condividono o meno i suggerimenti proposti dagli altri utenti.

Analisi della concorrenza e delle best practice

Tra le attività svolte nella fase di esplorazione dei requisiti è di fondamentale importanza il confronto con prodotti concorrenti simili già presenti sul mercato per "attingere" dai ◀ best practice ▶ qualche indicazione preziosa nonché compararne i prezzi, i punti di forza e riconoscere i punti di debolezza.

◀ Best practice Il termine di best practice rientra nel più ampio processo di benchmarking: metodo di confronto finalizzato a identificare, comprendere e adattare pratiche particolarmente significative (riconosciute come best practice o high performance) messe in atto da altre organizzazioni, al fine di migliorare le prestazioni della propria attività. ▶

Questa operazione è particolarmente difficile da effettuarsi nelle situazioni di sviluppo ad hoc altamente specializzato dove è difficile, se non impossibile, trovare altre realizzazioni già implementate: in generale, però, parte del sistema da sviluppare lo si può trovare già realizzato e quindi da queste realizzazioni si possono trarre preziose indicazioni.

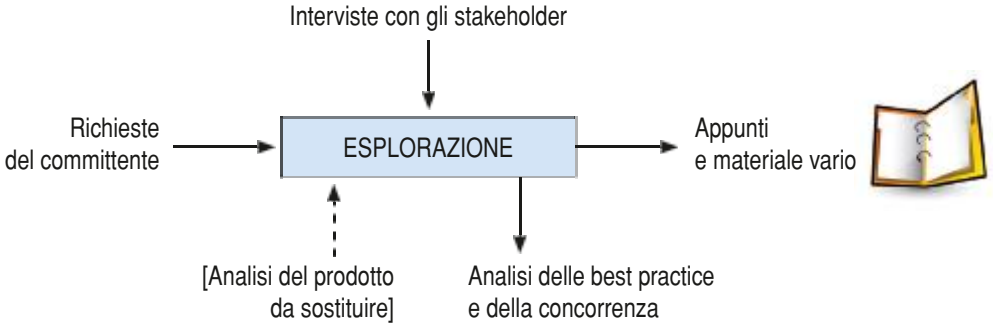
La ricerca di soluzioni già esistenti o simili deve essere fatta all'inizio del progetto, non appena sono definiti gli obiettivi essenziali; è importante anche sottoporre le soluzioni adottate da terzi agli stakeholder in modo che possano valutarle e commentarle per verificare la loro possibile integrazione nel nuovo sistema e la loro applicabilità nel contesto corrente.

Casi d'uso

Sono la descrizione degli esempi d'uso del sistema, cioè delle singole operazioni e attività che devono essere implementate nel sistema; per ciascun caso d'uso viene specificata la normale sequenza di eventi necessaria per compiere quell'operazione e le eventuali sequenze di eventi alternative in caso di errore per le situazioni non previste o per i "casi limite di funzionamento".

Per la loro importanza, che non si limita alla fase di analisi dato che i casi d'uso sono una componente fondamentale dalla fase di verifica e collaudo del sistema, verranno trattati dettagliatamente nella prossima lezione, a loro riservata.

Possiamo schematizzare con la seguente figura la fase di esplorazione, dove tutto il materiale raccolto è ancora allo stato di *appunti* e deve essere formalizzato nel documento SRS.

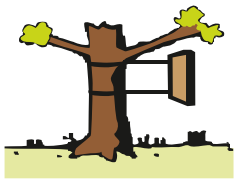


Problemi della fase di esplorazione

La metafora più comune per la gestione dei requisiti nei progetti software è quella conosciuta come la best practice metafora dell'altalena, che aiuta a comprendere perché le parole (influenzate dal gergo) non sono indicate per descrivere il problema.

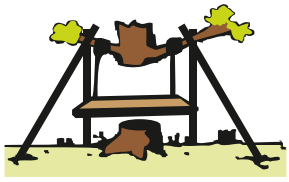
Quello che ha chiesto il cliente

Quello che ha capito l'analista



Come è stato tradotto in fase di progetto

Prototipo costruito in fase di realizzazione



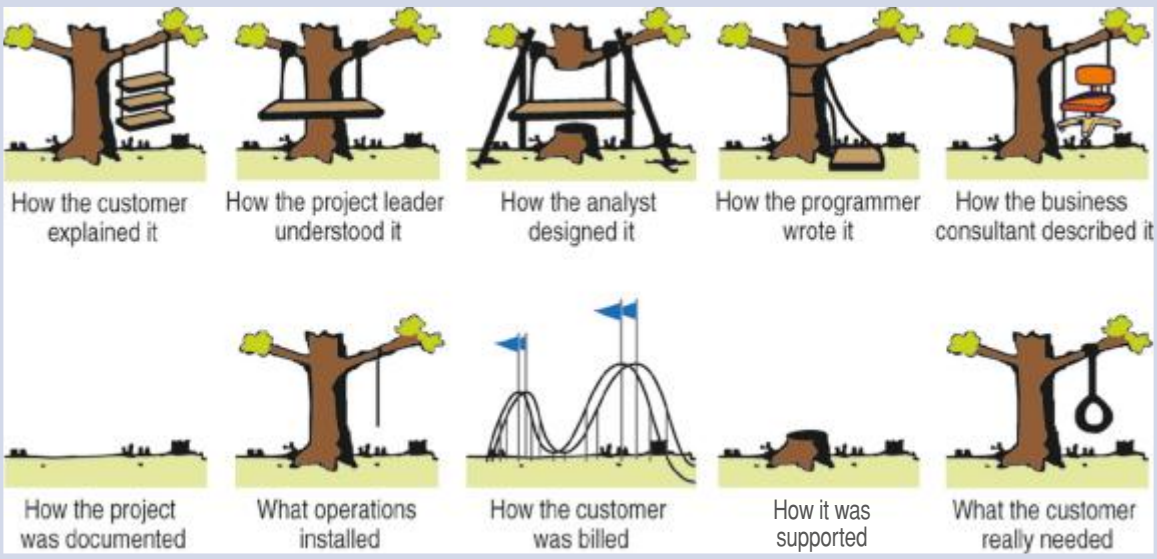
Modifiche apportate in fase di test e debug

Esigenza reale che aveva il cliente

È chiaro come l'esigenza descritta dal cliente "con sue parole" venga interpretata diversamente da tutti gli interlocutori nelle diverse fasi del ciclo di vita dello sviluppo del prodotto: è interessante infine osservare come lo stesso cliente *abbia una diversa percezione* di quello che realmente gli serve rispetto a quello di cui "dichiara di avere bisogno".

> La difficoltà di comprensione delle esigenze del cliente rende difficile la realizzazione del sistema.

◀ Riportiamo per completezza una nuova formulazione più completa della "*metafora dell'altalena*". ▶



Un'altra metafora che ben si presta alla gestione dei requisiti è il "*telefono senza fili*", un gioco diffuso tra i bambini "*prima che si diffondessero i cellulari*" (il gioco forse si gioca ancora, ma probabilmente avrà cambiato nome).
Il gioco è semplice: i bambini si dispongono in linea oppure in cerchio; il primo sussurra una frase nell'orecchio del vicino, che a sua volta la ripete al vicino successivo, e così via fino all'ultimo bambino che, per concludere, deve dire la frase ad alta voce. Più sono i bambini coinvolti nella sequenza, più la frase finale risulta diversa dalla frase iniziale.

La gestione dei requisiti è analoga: più lunga è la catena che collega chi ha l'esigenza, il requisito, a chi lo deve soddisfare, maggiore la probabilità di fraintendimenti e incomprensioni. Più persone lavorano in sequenza sui requisiti, più aumentano i costi, i tempi, i rischi, la litigiosità.

La fase di esplorazione è estremamente delicata e, come abbiamo visto, non sempre è di semplice realizzazione in quanto spesso presenta notevoli difficoltà, che possono essere suddividere in quattro tipologie:

- problemi di ambito: durante le interviste è difficile individuare il livello di dettaglio e di approfondimento che deve essere rispettato, rischiando da una parte di non entrare abbastanza nello specifico e tralasciare magari aspetti e dettagli che invece sono di importanza essenziale nell'impostazione del sistema, oppure di "sconfinare" in aree esterne a quelle specifiche di competenza dello stakeholder interrogato delle quali ha solo una conoscenza parziale e quindi potrebbe esprimere valutazioni dannose;
- problemi di comprensione: la difficoltà di comunicazione è stata già più volte sottolineata ed è dovuta dal fatto che gli stakeholder utilizzano un linguaggio tecnico differente da chi raccoglie i requisiti e tende a non dare il giusto peso e la corretta importanza alle attività che svolgono, trascurando gli aspetti che ritengono ovvi e limitandosi a descriverle superficialmente. Va sempre tenuto presente che gli stakeholder non sono esperti di informatica e spesso le loro richieste sono di difficile realizzazione oppure potrebbero avere costi elevati non contemplati dal budget del cliente;
- problemi di conflitto: il medesimo requisito può essere descritto in modo differente da stakeholder che appartengono a gruppi diversi e, al limite, le descrizioni potrebbero essere anche incompatibili tra di loro. I conflitti devono emergere in questa fase e devono essere affrontati in modo da trovare una "mediazione" che soddisfi tutti i contendenti;
- problemi di volatilità: i requisiti individuati non rimangono stabili per tutto il ciclo di vita del progetto, anzi, è anche possibile che si evolvano e mutino anche all'interno della singola fase di analisi. Le cause principali di questa dinamicità sono legate a eventi esterni come l'evoluzione tecnologica, i mercati, le leggi e gli obblighi fiscali, oppure a eventi interni, come il ricambio del management o la ristrutturazione dell'organizzazione del committente, acquisizioni o vendite societarie, nuove strategie di mercato, di produzione ecc.

È difficile stabilire la correttezza e la completezza dei requisiti, specialmente prima che il sistema venga realizzato e quindi collaudato: dato che la specifica dei requisiti serve come base per lo sviluppo, questi devono essere rivisti con grande attenzione sia dal committente che dal team di progettazione e validati prima che si prosegua con le successive fasi di realizzazione.

Verifichiamo le conoscenze

Esercizi a scelta multipla

1 Quale tra le seguenti non è una attività della ingegneria dei requisiti?

a) raccolta dei requisiti
b) analisi dei requisiti
c) stesura della documentazione dei requisiti SRS
d) collaudo dei requisiti
e) approvazione dei requisiti

2 Quale tra le seguenti non è una tipologia di progetto?

a) greenfield engineering
b) re-engineering
c) reverce engineering
d) interface engineering

3 Quale tra le seguenti tecniche di esplorazione fa emergere aree di conflitto?

a) Interviste individuali
b) Focus group
c) Osservazioni sul campo
d) Suggerimenti spontanei degli utenti
e) Questionari
f) Analisi della concorrenza e delle best practices
g) Scenari e casi d'uso

4 Quali tra le seguenti tecniche di esplorazione richiedono molto tempo?

a) Interviste individuali
b) Focus group
c) Osservazioni sul campo
d) Suggerimenti spontanei degli utenti
e) Questionari
f) Analisi della concorrenza e delle best practices
g) Scenari e casi d'uso

5 Quale tra i seguenti problemi non è connesso alla fase di esplorazione?

a) problemi di ambito
b) problemi di comprensione
c) problemi di sviluppo
d) problemi di conflitto
e) problemi di volatilità

Test vero/falso

1	Se esiste già nella azienda un sistema che deve essere riprogettato si parla di re-engineering.	V F
2	Il termine legacy è utilizzato per indicare qualcosa di valore ottenuto attraverso un'eredità.	V F
3	Nell'ambito dei sistemi informativi il termine legacy sta a indicare qualcosa di vecchio.	V F
4	Con Stakeholder Engagement si intende la scelta degli attori.	V F
5	La raccolta di informazioni dagli stakeholder può essere fatta sotto forma di sondaggio.	V F
6	La migliore fonte per reperire i requisiti è quella della interviste individuali.	V F
7	Agli stakeholder è necessario effettuare interviste strutturate.	V F
8	L'attendibilità delle risposte ai questionari è generalmente buona.	V F
9	I requisiti individuati non rimangono stabili per tutto il ciclo di vita del progetto.	V F
10	I requisiti individuati non possono variare nella fase di analisi.	V F

LEZIONE 3

SCENARI E CASI D'USO

IN QUESTA LEZIONE IMPAREREMO...

- a individuare gli scenari d'uso
- a descrivere i casi d'uso in UML
- a documentare i casi d'uso

Scenari d'uso e casi d'uso

Una tecnica fondamentale nella realizzazione di un nuovo prodotto è quella di individuare gli scenari d'uso e i casi d'uso.

SCENARIO

Uno ◀ **scenario** ▶ è una descrizione concreta, focalizzata, informale di una **singola caratteristica** del sistema utilizzata da un singolo attore.

◀ **Scenario** "A narrative description of what people do and experience as they try to make use of computer systems and applications" [M. Carrol, Scenario-based Design, Wiley, 1995] ▶

Quindi uno scenario d'uso è la descrizione che uno specifico attore fornisce, in linguaggio comune, del suo utilizzo del sistema, o meglio, della "sequenza di operazioni" che egli effettua per una possibile situazione di utilizzo del sistema.

ESEMPIO 7 ***Un semplice scenario: negozio online***

Vediamo per esempio uno scenario riferito a un *negozio online* su Internet.

Scenario: *Acquisto di un Prodotto*

Il cliente naviga nel catalogo e raccoglie gli articoli desiderati in un carrello della spesa "virtuale". Quando il cliente desidera pagare, descrive la modalità di spedizione e fornisce la necessaria

informazione riguardante la propria carta di credito prima di confermare l'acquisto. Il sistema controlla se la carta di credito è valida e conferma l'acquisto sia immediatamente che con un successivo messaggio di posta elettronica.

Questo scenario descrive come potrebbero andare le cose, ma se per esempio la carta di credito fosse scaduta la parte terminale dello scenario sarebbe diversa:
…

Il sistema controlla se la carta di credito è valida ed essendo scaduta invia un messaggio di posta elettronica segnalando il problema al cliente.

Potrebbe esserci un altro problema sempre legato al pagamento, per esempio se il credito non fosse sufficiente, modificando nuovamente la parte terminale dello scenario:
…

Il sistema controlla se la carta di credito è valida ed essendo il credito insufficiente invia un messaggio di posta elettronica segnalando il problema al cliente.

Uno scenario può anche essere descritto mediante un "dialogo" che rappresenta l'interazione tra gli utilizzatori e il sistema:
- il cliente richiede l'elenco dei prodotti
- il sistema propone i prodotti disponibili
- il cliente sceglie i prodotti che desidera
- il sistema fornisce il costo totale dei prodotti selezionati
- il cliente conferma l'ordine
- il sistema comunica l'accettazione dell'ordine

È importante focalizzare l'attenzione sulle **funzionalità generate dalla interazione** e non dalle attività oppure da come queste vengono svolte all'interno del sistema.

CASO D'USO

Un **caso d'uso** è un insieme di scenari legati da un obiettivo comune per l'attore.

ESEMPIO 8 ***Negozio online – seguito***

Considerando l'esempio precedente del *negozio online*, il caso d'uso *Acquisto di un Prodotto* ha i seguenti tre scenari corrispondenti a:
- successo della transazione;
- invalidità della carta di credito;
- insufficienza del credito sulla carta.

Tipi di scenari

Esistono diversi tipi di scenari:

- As-is scenario: serve per descrivere la situazione attuale esistente nell'organizzazione e solitamente viene utilizzato durante il re-engineering: agli utenti viene detto di descrivere il sistema che stanno utilizzando;
- Visionary scenario: viene usato principalmente dal committente tipicamente nelle situazioni di greenfield engineering o nel re-engineering per descrivere il sistema futuro o desiderato;
- Evaluation scenario: descrive i task che gli utenti dovrebbero svolgere nel sistema per i quali sarà poi valutato (esempio: emettere un biglietto, prenotare un posto, validare un pagamento ecc.);
- Training scenario: è un tutorial che indica passo per passo come un neofita può imparare la corretta interazione con il sistema (esempio: i passi necessari per emettere un biglietto: selezionare lo spettacolo, scegliere l'orario, cercare il posto, cercare la tariffa ecc.).

La definizione degli scenari può essere il primo passo nella definizione dei requisiti che avvicina i progettisti e gli sviluppatori agli stakeholder: la descrizione delle situazioni d'uso concrete è la base su cui lavorare per sviluppare il nuovo prodotto che dovrà soddisfare le esigenze reali, tipiche delle attività di ogni giorno.
La descrizione dettagliata degli scenari con esempi pratici aiuta ogni componente del gruppo di lavoro a comprendere senza pericolo di ambiguità le funzionalità desiderate.

> Gli scenari possono essere utilizzati in diverse attività del ciclo di vita del software: come vedremo i casi d'uso saranno fondamentali anche nella fase di verifica e collaudo in quanto il sistema deve implementare tutti i casi d'uso descritti negli scenari.

Individuazione degli scenari

L'individuazione degli scenari è sicuramente più difficile da effettuarsi nel caso che il sistema sia da realizzarsi ex novo, cioè se non esiste nella organizzazione: nel caso di greenfield engineering non bisogna aspettarsi che il cliente proponga e descriva gli scenari ma questa operazione deve essere fatta in modo coordinato tra team di sviluppo e cliente usando tecniche evolutive e incrementali.

Si può partire con semplici domande del tipo:

- quali sono i compiti primari che ciascun attore vuole che esegua il sistema?
- quali dati saranno creati, memorizzati, modificati, cancellati, o aggiunti dall'utente nel sistema?
- di quali cambiamenti esterni l'attore deve informare il sistema? Quanto spesso? Quando?
- di quali cambiamenti o eventi deve essere informato l'attore dal sistema? Entro quanto?

Quindi si riesamina assieme al cliente e agli stakeholder quanto è emerso da una "prima analisi" e si procede successivamente mediante affinamenti successivi: la continua interazione con il cliente aiuta gli analisti che man mano descrivono gli scenari acquisiscono una maggiore conoscenza dei requisiti.

L'individuazione degli scenari è più agevole se il sistema già esiste, cioè nelle situazioni di interface engineering o reengineering: in questo caso si parte da situazioni esistenti, ed è però necessario insistere sull'osservazione dettagliata delle funzioni per non ignorare particolari che gli stakeholder considerano scontati perché consueti e spesso eseguiti in automatico ma che sono ignoti agli sviluppatori.

I casi d'uso

Un caso d'uso (*use case*) specifica tutti i possibili scenari per una data funzionalità.
Quindi in un caso d'uso devono essere contemplate tutte le possibili alternative e i possibili sviluppi (scenari) di una specifica funzionalità: quindi è generalmente composto da un insieme di intera-

zioni, ciascuna di esse iniziata da un attore, e nella sua evoluzione può coinvolgere o meno altri attori.
Al caso d'uso viene dato un nome che di solito è composto da un verbo alla terza persona singolare o all'infinito e da un complemento, oppure da una breve frase che descrive sinteticamente lo scopo dell'interazione.

SCENARIO
Uno scenario è un'*istanza* di un caso d'uso.

ESEMPIO 9

Riprendiamo come esempio la situazione del *negozio online* ed elenchiamo alcuni possibili casi d'uso:
- registrare un nuovo utente
- modificare i dati di un utente
- acquistare un prodotto
- ricercare un prodotto nel catalogo
- inserire un nuovo prodotto in catalogo
- modificare i dati di un prodotto

…

Ciascuno di questi casi d'uso corrisponde a una precisa funzionalità che un "soggetto" deve eseguire con il nuovo sistema: possiamo individuare in questo esempio quattro diversi attori che devono interagire con il **sistema software**, cioè l'*Utente Visitatore*, l'*Utente registrato*, l'*Amministratore del sistema* e il *Sistema Bancario* e ciascuno di questi attori invocherà specifici casi d'uso, anche comuni a più attori.

ESEMPIO 10

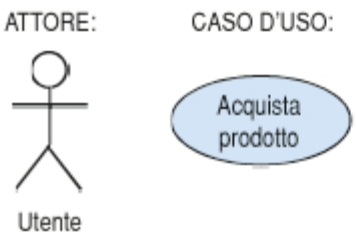
In particolare nel nostro esempio avremo:
- **A** l'*Utente Visitatore*
  - ricercare un prodotto nel catalogo
  - registrare un nuovo utente
- **B** l'*Utente registrato*
  - modificare i dati di un utente
  - ricercare un prodotto nel catalogo
  - acquistare un prodotto
- **C** l'*Amministratore del sistema*
  - inserire un nuovo prodotto in catalogo
  - modificare i dati di un prodotto
- **D** *Sistema Bancario*
  - acquistare un prodotto

Se un caso d'uso coinvolge più attori, quello che persegue lo scopo che il caso d'uso deve soddisfare sarà considerato l'*attore principale*: nel nostro esempio nel caso *acquistare un prodotto* l'Utente Registrato è l'*attore principale* mentre il Sistema Bancario è l'*attore secondario*.

Descrizione del caso d'uso

Utilizziamo i diagrammi UML per la rappresentazione del caso d'uso: con questi possiamo descrivere i singoli scenari e anche indicare le relazioni esistenti tra essi.

Gli attori sono rappresentati con omini stilizzati e il nome dell'attore viene indicato sotto l'omino, i casi d'uso con ellissi e il loro nome è indicato all'interno dell'ellisse, come indicato nella seguente figura:



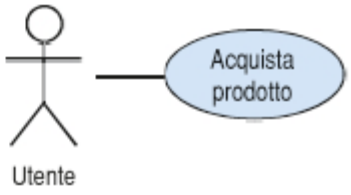
Anche se vengono indicati graficamente con degli omini, gli attori coinvolti in un caso d'uso possono anche non essere umani: per esempio, se un sistema si interfaccia con un altro mediante una linea telefonica per comunicare o ricevere dati oppure per richiedere un servizio non necessariamente è necessaria la presenza umana.
In questo caso nel diagramma al posto del nome dell'attore viene genericamente indicato **<sistema>** per indicare che l'attore non è umano.

ESEMPIO 11

Nel nostro esempio una situazione di questo tipo sarà sicuramente presente in quanto avverrà una richiesta da parte del nostro sistema alla banca per verificare la situazione della carta di credito.

Tra un attore e un caso d'uso viene indicata l'associazione mediante un segmento che li congiunge: in questo modo viene indicata una (o più) delle tre possibili situazioni:

- l'attore esegue il caso d'uso;
- l'attore fornisce informazioni al caso d'uso;
- l'attore riceve informazioni dal caso d'uso.



Si è quindi ottenuto un componente dell'Use Case Diagram (UCD): l'Use Case Diagram mostra come i casi d'uso sono collegati tra loro e con gli attori.

ESEMPIO 12

Completiamo l'Use Case Diagram del nostro esempio *negozio online* con tutti i casi d'uso che abbiamo prima individuato: ▶

La direzione della freccia non specifica la direzione dei flusso dei dati e viene messa solamente nei casi in cui possono esserci ambiguità nel caso di più attori per individuare l'attore principale.

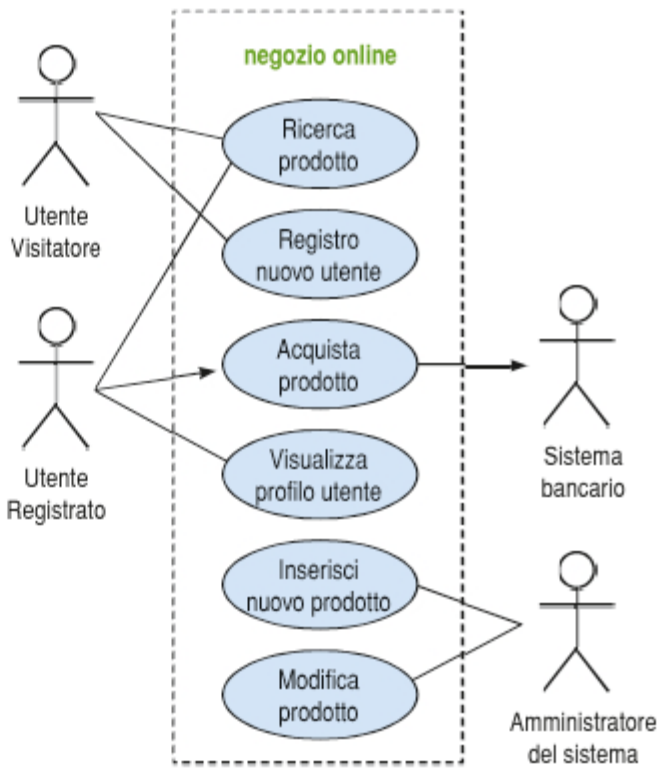


DIAGRAMMA DI CONTESTO

Un diagramma che rappresenta tutti i casi d'uso di un sistema si chiama diagramma di contesto del sistema, perché indica i "confini" dello stesso e tutti gli attori che lo utilizzano.
Il sistema viene indicato con un riquadro che contiene i casi d'uso e ne riporta il nome, come nel nostro esempio.

Vediamo un secondo esempio.

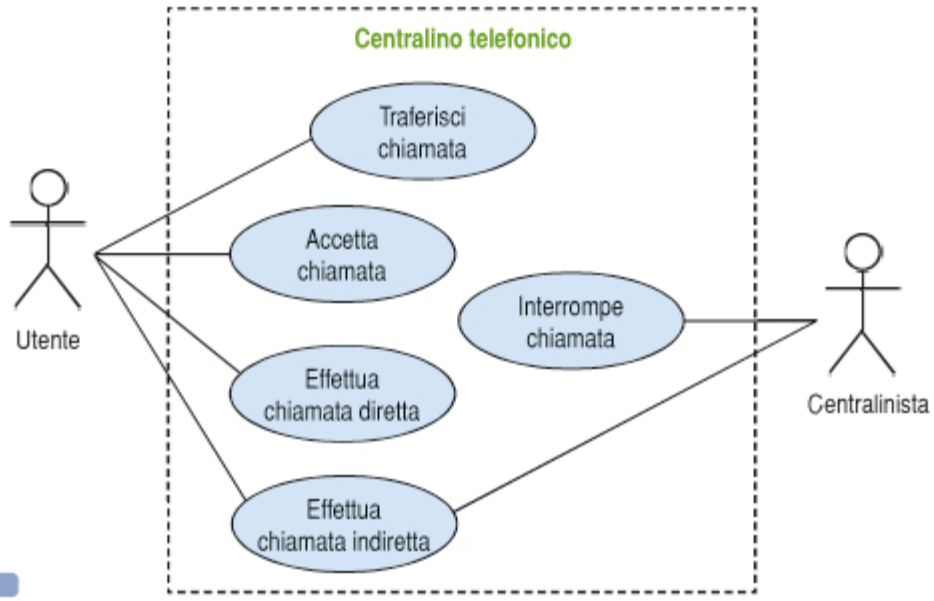
ESEMPIO 13 ***Centralino e chiamate telefoniche***

Si vuole realizzare un sistema di gestione delle chiamate telefoniche per un ufficio dove oltre alla linea diretta è presente anche un centralino per chi non conosce il numero degli interni.

Gli attori in questo sistema sono solamente due, gli *utenti* e la *centralinista*, ed elenchiamo i casi d'uso:

- l'utente effettua una chiamata diretta;
- la chiamata dell'utente viene accettata;
- la chiamata dell'utente viene trasferita;
- l'utente effettua una chiamata indiretta tramite centralinista;
- il centralinista interrompe una chiamata.

L'Use Case Diagram è il seguente:



Relazioni tra use case

Anche nella stesura degli *use case* si applica la metodologia top-down, partendo da un elenco dove la descrizione è generica e ad alto livello per poi raggiungere mediante affinamenti (e decomposizioni) un livello di dettaglio soddisfacente.

È anche possibile individuare delle relazioni tra *use case* e descriverle in modalità grafico nei diagrammi UML.

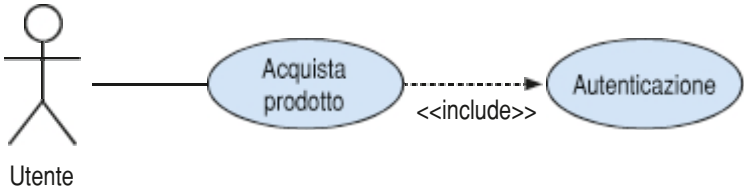
A È possibile riutilizzare gli use case, secondo due modalità.

1 **Inclusione (inclusion)**: questa modalità permette di utilizzare i passi appartenenti a una sequenza di un *use case* e inglobarli in un altro *use case*, cioè serve a rappresentare un caso d'uso che ne utilizza un altro.
Per rappresentare graficamente l'inclusione si traccia una freccia tratteggiata che punta sul caso da cui dipende l'altro caso, con la etichetta <<include>>.

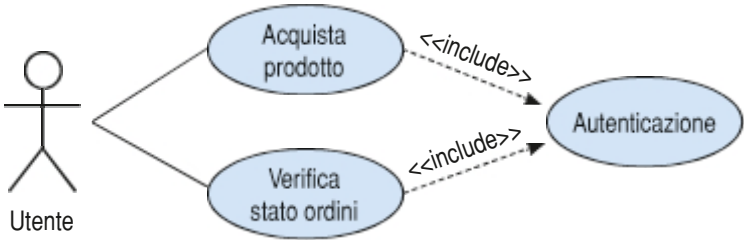
> La parola **include** racchiusa tra doppie parentesi formate dai simboli "<<" e ">>" prende il nome di stereotipo: <<include>> è uno stereotipo.

ESEMPIO 14

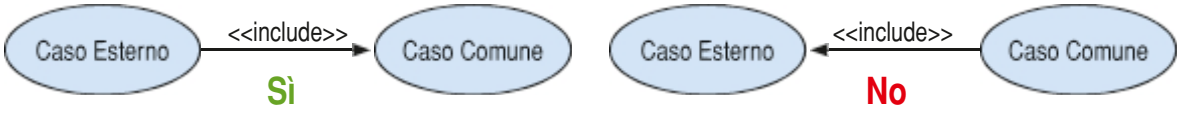
Nel nostro esempio abbiamo indicato genericamente la funzione *Acquista Prodotto* con la sequenza di operazioni che permettono di effettuare l'acquisto: questa attività è scomponibile in diverse parti, tra le quali sicuramente è presente la fase di *Autenticazione*, fase che potrebbe anche essere utilizzata in un ulteriore caso d'uso come la *Verifica dello stato dell'ordine* (da noi non considerata): quindi Autenticazione sarà indicata separatamente in un altro *caso d'uso* a se stante e nella descrizione dei casi d'uso che lo includono viene indicato col seguente diagramma:



Se ora aggiungiamo anche il caso d'uso ***Verifica** dello stato dell'ordine* otteniamo:



Un errore tipico è nel posizionamento della freccia, che deve andare dal caso più esterno al caso comune:

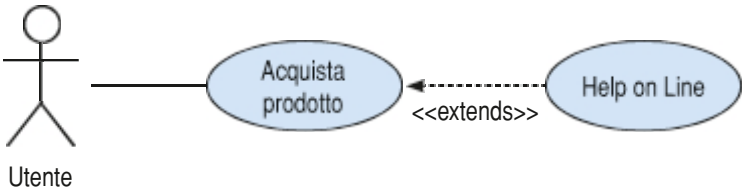


2 **Estensione (extension)**: tramite questo metodo è possibile creare un nuovo *use case* semplicemente aggiungendo dei passi a un *use case* esistente: in questo modo è possibile integrare l'*use case* aggiungendo comportamenti alternativi o eccezionali (opzionali) che estendono il caso d'uso generale.

Anche in questo caso si utilizza una linea tratteggiata con una freccia finale insieme con uno stereotipo che mostra la parola <<extends>> tra parentesi.
La relazione extends va utilizzata quando abbiamo uno *use case* simile a un altro che però fa qualcosa in più.

ESEMPIO 15

Aggiungiamo per esempio al nostro *NegozioOnline* un *Help on Line* che, per esempio, può essere richiamato durante l'acquisto di un prodotto: la situazione nel diagramma è la seguente

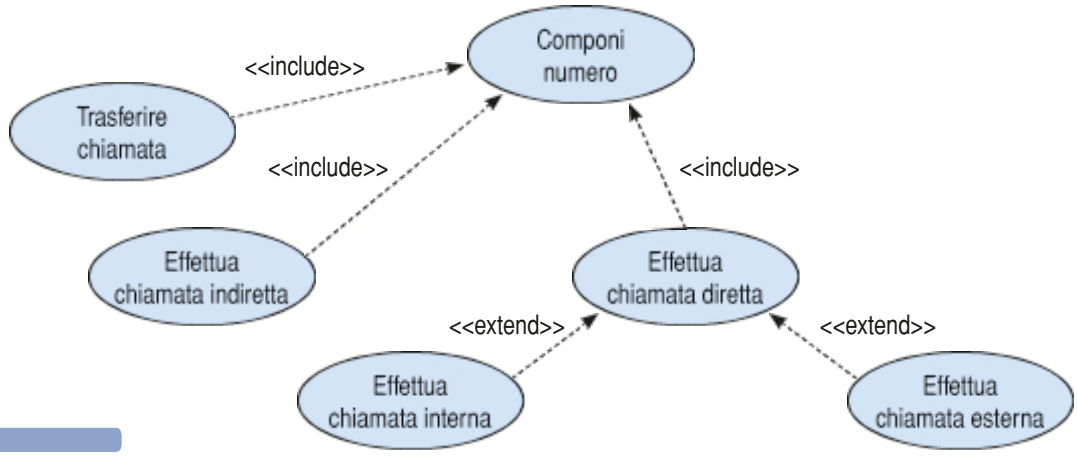


Anche in questo caso un errore tipico è nel posizionamento della freccia, che deve andare dal caso variante al caso standard:

È bene osservare una semplice regola per effettuare un utilizzo corretto (e moderato!) delle **inclusion** ed **extension**:
- utilizzare **includes** quando si sta ripetendo la descrizione di un comportamento presente già in uno o più differenti *use case* e il suo utilizzo è sempre richiesto;
- utilizzare **extends** quando si descrive una variazione al comportamento normale (utilizzo opzionale).

ESEMPIO 16 ***Esempio centralino telefonico – seguito***

Se rianalizziamo l'esempio del centralino telefonico potremmo descrivere le attività in termini di inclusione ed estensione come di seguito riportato, dove l'unica attività è quella di effettuare la chiamata e le altre attività sono "varianti" o "possibilità" della stessa operazione:



B Generalizzazione (generalization)

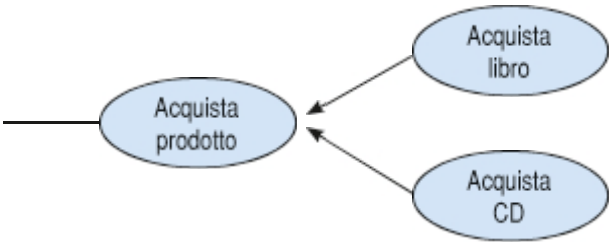
Un *use case* può inoltre ereditare le caratteristiche di un altro *use case* (come nel caso di ereditarietà delle classi nella OOP): lo *use case* figlio eredita il comportamento e il significato dal padre e in più aggiunge le sue caratteristiche specifiche.

> Come per le classi la relazione di ereditarietà deve soddisfare la relazione IS-A, nel caso di *use case* è possibile applicare lo *use case* figlio dove è possibile applicare il padre.

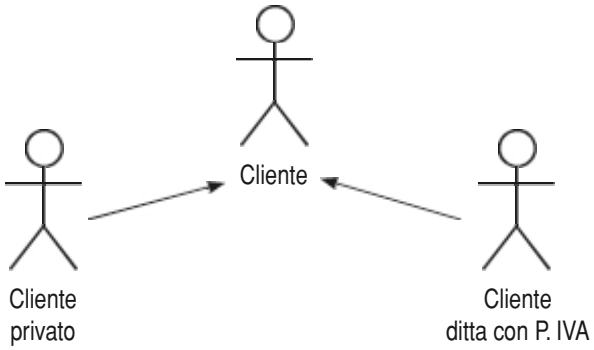
La generalizzazione viene rappresentata con una linea continua che ha un triangolo aperto che parte dai figli e punta al padre.

ESEMPIO 17 ***Negozio online – seguito***

Nell'esempio del *NegozioOnline* potrei avere due tipi di prodotti, libri e CD: per descrivere che possono essere entrambi delle "specializzazioni" della funzione *Acquista prodotto* (che è il caso generale) utilizzo la seguente notazione: ▶



La relazione di generalizzazione può esistere anche tra actors: sempre nell'esempio del *NegozioOnline* potrei avere due tipi di clienti, gli utenti privati o le ditte (con partita IVA), che il sistema deve trattare in modo differente (scontrino fiscale per i primi e fattura per i secondi); li rappresento col seguente diagramma: ▶



Documentazione dei casi d'uso

La descrizione dei casi d'uso del sistema costituisce un capitolo molto importante del documento di specifica dei requisiti: a ogni caso d'uso mostrato nel diagramma deve essere associata una descrizione per renderlo comprensibile a tutti coloro che lo devono poi leggere e verificare, tra i quali sono presenti gli stakeholder che non hanno conoscenze informatiche.
Quindi si utilizza il linguaggio naturale in modo semplice e chiaro, in modo da non creare ambiguità.
Lo schema adottato da tutti i progettisti è quello riportato di seguito, dove in ogni caso d'uso si descrivono tutti gli scenari, a partire da quello principale che è quello che accade nella grande maggioranza dei casi, detto basic flow o "happy path":

Nome del Caso d'uso
Descrizione: <sintesi della attività>
Attori: <descrizione degli attori coinvolti nel caso d'uso>

Scenario principale

- **Entry condition** (precondizione)**:** è un vincolo che il sistema deve rispettare affinché il caso d'uso possa iniziare; si incomincia con "Questo caso d'uso inizia quando…";

- **Flusso di Eventi:** <descrizione in linguaggio naturale informale>;
- **Exit condition** (postcondizione): è una condizione che è verificata quando il caso d'uso termina (può essere diversa a seconda dello scenario effettivamente seguito): si incomincia con "Questo caso d'uso termina quando…";

Scenari alternativi
- **Flussi alternativi:** < descrizione delle modalità alternative di esecuzione del caso>
- **Eccezioni**: < descrizione di cosa accade in tutte le situazioni di errore, cioè quando non va a buon fine il caso principale>

Requisiti Speciali: sono eventuali requisiti non funzionali (cioè non relativi alle funzionalità del sistema nell'assolvere al caso d'uso) e i vincoli;
Extension Points: sono relazioni con eventuali altri casi d'uso correlati;
Frequenza stimata di utilizzo, per decidere le priorità nel piano di sviluppo;
Criticità: per stimare il rischio legato al requisito.

Lo schema descritto è una interpretazione di quello specificato da Ivar Jacobson, ideatore assieme a Grady Booch e James Rumbaugh dell'UML.

ESEMPIO 18 *Descrizione degli scenari del Negozio Online*

Vediamo sempre nell'esempio NegozioOnline che come scenario principale abbiamo il successo dell'acquisto mentre gli scenari indesiderati (alternative flows) possono per esempio essere:
- carta di credito non accettata;
- disponibilità di credito non sufficiente;
- collegamento con i servizi interbancari interrotto;
- disponibilità di un prodotto terminata.

Attori: Utente, Sistema Bancario

Scenario principale

- **Entry condition**: per poter fare un acquisto on-line l'utente deve essere registrato;
- **Flusso di Eventi**:
  1. Il cliente *ricerca nel catalogo* e inserisce nel carrello uno o più articoli
  2. Il cliente va "alla cassa"
  3. Il sistema presenta il conto degli articoli selezionati
  4. Il cliente inserisce le informazioni per la spedizione (indirizzo, tempo di consegna)
  5. Il sistema fornisce il conto totale, comprese le spese di spedizione
  6. Il cliente inserisce le informazioni riguardo la sua carta di credito
  7. Il sistema autorizza l'acquisto
  8. Il sistema conferma il perfezionamento con successo dell'ordine
  9. Il sistema invia una e-mail di conferma dell'acquisto all'indirizzo indicato dal cliente
- **Exit condition**: se il cliente conferma l'ordine esso viene passato al magazzino, se invece l'utente lo annulla il sistema rimane inalterato;

Scenari alternativi
Eccezione a) Carta di credito non valida
- Il sistema al passo 7 del Basic Flow non autorizza l'acquisto.
- Il sistema avverte l'utente e gli consente di reinserire i dati

Eccezione b) Carta di credito non valida
- Il sistema al passo 7 del Basic Flow non autorizza l'acquisto.
- Il sistema avverte l'utente del credito esaurito e gli consente di reinserire i dati di una diversa carta di credito.

Eccezione c) Collegamento con i servizi interbancari interrotto
- Il sistema salva il carrello dell'utente
- lo invita a riprovare a perfezionare l'ordine in seguito

Eccezione d) <altre situazioni>

Requisiti Speciali
- Il sistema deve garantire che l'inoltro dell'ordine al magazzino avvenga entro le 24 ore successive alla sua conferma.
- Gli articoli richiesti dagli utenti devono essere presenti in magazzino almeno il 90% delle volte

Se un caso d'uso ne richiama un altro, il nome di quest'ultimo viene sottolineato (si indica con la sottolineatura perché suggerisce un collegamento ipertestuale): nel nostro esempio si può vedere che il primo passo ha *ricerca nel catalogo* che è un caso d'uso descritto in un altro documento (.si dice allora che questo caso d'uso è *incluso* nel precedente).

Ogni passo di uno scenario viene espresso con una frase semplice, che esprime chiaramente la funzione svolta, senza entrare nei dettagli di **come** viene svolta: testi molto lunghi rischiano di non essere letti.

◀ **Alistair Cockburn** ▶, autore di un libro sui casi d'uso, dà le seguenti indicazioni:

Molti si sentono colpevoli se lo scenario principale di un caso d'uso è breve, così lo allungano per arrivare a quindici, o anche trentacinque righe. Personalmente, io non ho mai scritto uno scenario principale più lungo di nove passi. Non che il nove sia un numero magico; il fatto è che, quando ho individuato i sotto-goal a un giusto livello e ho eliminato i dettagli che riguardano la progettazione, mi restano sempre meno di nove passi. A volte, lo scenario principale di un caso d'uso può essere anche di soli tre passi.
Il valore maggiore di un caso d'uso non è nello scenario principale, ma nei comportamenti alternativi. Lo scenario principale può occupare da un quarto a un decimo della lunghezza totale di un caso d'uso, perché ci possono essere molte alternative da descrivere. Se lo scenario principale fosse lungo trentacinque passi, l'intero caso d'uso occuperebbe dieci pagine, e sarebbe troppo lungo da leggere e da comprendere. Se lo scenario principale contiene da tre a nove passi, la descrizione complessiva potrebbe essere di solo due o tre pagine, il che è più che sufficiente.
Se potete evitare di includere troppi dettagli dell'interfaccia utente, i casi d'uso saranno molto più facili da leggere. E i casi d'uso leggibili possono in effetti venire letti. Casi d'uso lunghi e illeggibili vengono soltanto firmati – di solito con sgradevoli conseguenze sul progetto, alcuni mesi più tardi.

◀ **Alistair Cockburn** is a Senior Consultant with Cutter Consortium's Agile Product & Project Management practice. He is consulting fellow at Humans and Technology, where he helps clients succeed with object-oriented projects, including corporate strategy, project setup, staff mentoring, process development, technical design, and design quality. ▶

Concludiamo riportando alcuni semplici consigli che possono aiutare a realizzare dei buoni *use case diagram* e a documentarli correttamente:

- i nomi dei casi d'uso dovrebbero includere verbi;
- i nomi di attori dovrebbe essere sostantivi;
- i confini del sistema dovrebbero essere chiari: chiara distinzione delle azioni svolte dall'attore e delle azioni svolte dal sistema;
- le relazioni causali tra passi successivi dovrebbero essere chiare;
- un caso d'uso dovrebbe descrivere una transazione utente completa;
- le eccezioni dovrebbero essere descritte separatamente;
- un caso d'uso non dovrebbe descrivere un'interfaccia del sistema;
- un caso d'uso non dovrebbe superare due o tre pagine.

Ricordiamo inoltre che gli use case non servono solo nell'analisi dei requisiti ma vengono utilizzati nella validazione delle specifiche, nella stesura della documentazione, nella fase di codifica e infine per ricavare i dati di test con cui convalidare il sistema, dato che ogni use case rappresenta una funzionalità che deve essere presente e della quale deve essere verificata la correttezza.

Verifichiamo le competenze

Per ciascuna situazione descritta in seguito viene richiesto di:

1. definire gli attori
2. definire gli use case, diagramma e documentazione, aiutandosi con il seguente elenco di domande:
   - chi sono gli attori principali e secondari
   - quali sono gli obiettivi degli attori
   - quali sono le precondizioni
   - quali sono le principali azioni richieste/svolte dagli attori
   - quali eccezioni vanno considerate nello svolgimento della storia
   - quali variazioni vanno considerate nelle interazioni con gli attori
   - quali sono le informazioni acquisite, prodotte e modificate dagli attori

1 Supermercato

Viene richiesto di sviluppare un prodotto per offrire ai clienti di un supermercato un sistema di vendita on-line con il quale i clienti facciano via web le stesse cose che fanno quando visitano il supermercato fisico.
Identifica gli attori, i casi d'uso e disegna il diagramma di contesto.
Dopo aver individuato il caso d'uso più importante, descrivilo completamente con lo schema di Jacobson.

2 Supermercato bis

In riferimento all'esercizio precedente relativo al supermercato, descrivi la specifica del caso d'uso 'aggiorna carrello', dove il cliente può eseguire due operazioni:

- aggiungere un nuovo prodotto al carrello;
- modificare la nuova quantità di un articolo esistente;
- rimuovere l'articolo dal carrello.

Descrivi la specifica di questo nuovo caso d'uso.

3 Supermercato ter

In riferimento all'esercizio precedente relativo al supermercato, viene introdotto un sistema di consegna a domicilio della spesa per pensionati.
Descrivi la specifica di questo nuovo caso d'uso.

4 Medico Generico

Un medico vuole realizzare un sistema per gestire i propri pazienti in modo da non dover chiedere al paziente la sua situazione e poter annotare di volta in volta cure, situazione, osservazioni ecc.
Identifica gli attori, i casi d'uso e disegna il diagramma di contesto.
Dopo individuato il caso d'uso più importante, descrivilo completamente con lo schema di Jacobson.

5 Medico Generico bis

In riferimento all'esercizio precedente relativo al medico, viene introdotto un sistema di prenotazione on-line degli appuntamenti.
Descrivi la specifica di questo nuovo caso d'uso.

6 Medico Generico ter

In riferimento all'esercizio precedente relativo al medico, viene introdotto un sistema di richiesta ricette ripetitive mediante e-mail.
Descrivi la specifica di questo nuovo caso d'uso.

7 Museo

I visitatori di un museo possono accedervi comprando un biglietto venduto da un addetto alla biglietteria o usando biglietti acquistati precedentemente e possono richiedere di essere accompagnati da una guida. È previsto uno sconto per alcune categorie di utenti, per i gruppi e le scolaresche.
Identifica gli attori, i casi d'uso e disegna il diagramma di contesto.
Dopo aver individuato il caso d'uso più importante, descrivilo completamente con lo schema di Jacobson.

8 Museo bis

In riferimento all'esercizio precedente relativo al museo, viene introdotto un sistema di prenotazione on-line per le scuole.
Descrivi la specifica di questo nuovo caso d'uso.

9 Biglietteria ferroviaria

Il sistema deve permettere ai viaggiatori di comprare i biglietti (andata e/o ritorno) e gli abbonamenti (settimanale, mensile, annuale); le tariffe vengono aggiornate da un sistema centralizzato.
Oltre alla biglietteria dove è presente un bigliettaio, è in funzione anche un distributore automatico di biglietti che però ha alcune possibili problematiche nell'emissione dei biglietti (mancanza di carta, di resto ecc.).
Identifica gli attori, i casi d'uso e disegna il diagramma di contesto.
Dopo aver individuato il caso d'uso più importante, descrivilo completamente con lo schema di Jacobson.

10 Biglietteria bis

In riferimento all'esercizio precedente relativo alla biglietteria, viene introdotto un sistema di prenotazione on-line per i biglietti di I classe e le tratte superiori a 50 km.
Descrivi la specifica di questo nuovo caso d'uso.

11 Biglietteria ter

In riferimento all'esercizio precedente relativo alla biglietteria, viene introdotto un sistema di sconti per le famiglie.
Descrivi la specifica di questo nuovo caso d'uso.

12 Sportello servizi comunali

Si vuole realizzare un sistema di sportello automatico per offrire alcuni servizi ai cittadini, come la stampa di certificati e il pagamento di tributi comunali (IMU, multe ecc.).
Il cittadino viene autenticato mediante la lettura della tessera sanitaria oppure con la digitazione del codice fiscale seguito dal PIN personale.
Identifica gli attori, i casi d'uso e disegna il diagramma di contesto.
Descrivi il caso d'uso della stampa certificati con lo schema di Jacobson.

13 Sportello servizi comunali bis

Descrivi gli scenari di successo e di insuccesso relativi al pagamento di una multa in contanti con errori dovuti al codice della multa non corretto, al time-out del sistema, all'annullamento esplicito dell'operazione, alla mancanza di fondi.
Descrivi la specifica di questo nuovo caso d'uso.

14 Sportello servizi comunali ter

Tra i documenti comunali è anche prevista la richiesta della cartografia tecnica regionale (CTR) nelle scale 1:5.000, 1:10.000 e 1:25.000 nei formati tradizionale cartaceo o elettronico relativa alla zona in cui il cittadino abita.
Nel caso di richiesta del formato elettronico, questa viene inviata immediatamente per e-mail all'utente mentre nel caso di richiesta cartacea viene plottata in un tempo successivo alla richiesta e comunicata all'utente sempre per e-mail la disponibilità della copia da ritirarsi presso l'ufficio tecnico.
La quota da pagare è diversa a seconda che la richiesta provenga da un privato cittadino o da un tecnico professionista.
Identifica gli attori, i casi d'uso e disegna il diagramma di contesto.
Dopo individuato il caso d'uso più importante, descrivilo completamente con lo schema di Jacobson.

LEZIONE 4

LA DOCUMENTAZIONE DEI REQUISITI

IN QUESTA LEZIONE IMPAREREMO...

- il documento di Specifica dei Requisiti Software (SRS)
- le caratteristiche e la struttura di un SRS
- la validazione e convalida delle specifiche di un SRS

Generalità

Il lavoro di individuazione dei requisiti termina con la stesura di un documento di Specifica dei Requisiti Software (SRS): questo è un documento ufficiale e descrive quello che è richiesto allo sviluppatore del sistema come risultato del lavoro di analisi tra il cliente, l'utente e lo sviluppatore.

È di fondamentale importanza per la buona riuscita dello sviluppo del sistema che l'SRS sia di alta qualità in quanto questo documento aiuta lo sviluppatore a comprendere il dominio di applicazione, riduce i tempi (e quindi i costi) di sviluppo e fornisce un punto di riferimento per la *convalida* del prodotto finale.

Un errore nell'SRS produrrà errori nel sistema finale: la rimozione di un difetto scoperto dopo lo sviluppo e rilascio del sistema può costare fino a 100 volte la rimozione di un difetto durante la fase di analisi e specifica dei requisiti.

Nel documento deve essere presente una analisi approfondita dell'utente, delle sue reali necessità, l'elenco dei singoli requisiti specificati e completati con i relativi casi d'uso.

Abbiamo quindi tre tipologie di informazioni che devono "emergere" nell'analisi:

- analisi degli utenti:
  - a quali categorie di utenti è destinato il prodotto?
  - quali sono le loro caratteristiche?
  - quali categorie vanno considerate prioritariamente?
- analisi dei bisogni:
  - quali sono le necessità di ciascuna categoria di utenti?
  - quali sono prioritari?

- analisi del contesto d'uso:
  – quali saranno i diversi contesti d'uso del prodotto da parte delle diverse categorie di utenti?
  – quali sono prioritari?

Nel documento dei requisiti (Requirements Documents), quindi, dopo una parte generale che descrive il sistema e le informazioni sulla richiesta del committente vengono raccolti in modo organizzato i requisiti individuati e su come ha avuto luogo la loro determinazione, e vengono allegati i diagrammi d'uso e le rispettive schede che li descrivono.

Per redigere un efficace documento SRS gli sviluppatori seguono i suggerimenti dello standard IEEE Std 830-1998 IEEE Recommended Practice for Software Requirements Specification.
Esistono anche altri modi per scrivere SRS di qualità, comunque lo Standard 830 è quello più diffuso ed è quello al quale anche noi faremo riferimento.

Requirements Documents proposto da Sommerville

Esistono diversi standard che propongono la forma del documento SRS e la metodologia di compilazione: noi descriveremo la proposta di Sommerville che si ispira allo standard IEEE/ANSI 830-1998.

1 Introduzione
1. Scopo del documento dei requisiti
2. Scopo del prodotto
3. Definizione, acronimi ed abbreviazioni (glossario)
4. Riferimenti
5. Overview dell'intero documento

2 Descrizione generale
1. Prospettive sul prodotto
2. Funzioni del prodotto
3. Caratteristiche degli utenti
4. Vincoli generali
5. Assunzioni e dipendenze

3 Requisiti specifici
1. Definizione dei Requisiti funzionali (requisiti utente)
2. Definizione dei Requisiti non funzionali (requisiti utente)
3. Architettura: strutturazione in sottosistemi a cui riferire i requisiti
4. Specifiche dei Requisiti di sistema
5. Modelli del sistema
6. Evoluzione del sistema

4 Appendici
1. Individuazione ed eventuale descrizione della piattaforma hardware
2. Requisiti di DataBase
3. Piani di Test

5 Indici

All'interno del documento vengono utilizzati i seguenti termini IEEE:
- contratto (*Contract*): il documento (legale) stilato tra committente e fornitore (la specifica dei requisiti potrebbe/dovrebbe essere la parte integrante del contratto);
- committente (*Customer*): colui che paga il prodotto che di solito (ma non necessariamente) è colui che decide i requisiti;

- fornitore (*Supplier*): chi produce il prodotto / sviluppa per il committente;
- utente (*User*): la persona che usa e interagisce direttamente col sistema (spesso si identifica col committente).

> Per ciascuna funzionalità descritta nella sezione 3.1 devono essere specificati gli input, l'output e l'azione elaborativa.
> Per le altre componenti del SRS non sono necessarie ulteriori spiegazioni.

Vediamo, in un esempio, come viene effettuata la descrizione di una singola specifica funzionalità (requisito utente) che, come più volte detto, deve rispondere alla seguente domanda: *Cosa deve fare il sistema software?*

ESEMPIO 19 *Negozio Online: funzionalità Inserimento articolo*

Descriviamo la funzionalità di inserimento di un articolo nel carrello del *NegozioOnline* descritto nelle precedenti lezioni:

3.x Inserisci articolo nel carrello
Utenti della funzionalità:
clienti registrati
Introduzione:
la funzionalità consente l'immissione di un nuovo articolo nel carrello di acquisto
Precondizione:
l'utente si è precedentemente autenticato come cliente registrato
Input:
codice articolo: obbligatorio
quantità: obbligatorio (di default proposto 1)
Elaborazione:
viene presentata una videata di ricerca degli articoli a più livelli e modalità
selezionato un articolo, si visualizza un form per l'immissione dei dati di input
viene effettuata la lettura dei dati di input
viene verificato che la quantità sia ammissibile
viene verificata la disponibilità dell'articolo e eventualmente segnalati i tempi di attesa
Output:
aggiunta di una riga nel carrello
segnalazione di errore nel caso di articolo già presente

Sempre del nostro NegozioOnline vediamo un altro esempio di funzionalità, quella relativa all'elenco dei clienti attivi.

ESEMPIO 20 *Negozio Online: funzionalità Elenco Clienti Attivi*

3.y Elenco Clienti Attivi
Utenti della funzionalità:
Titolare
Introduzione
La funzionalità visualizza la lista dei clienti che hanno effettuato acquisiti nell'ultimo anno con il totale degli acquisti effettuati e con possibilità di stampa.
Consente, se richiesto, di selezionare un cliente, di visualizzarne la scheda personale e l'elenco degli articoli acquistati ed eventualmente di stamparla.
Precondizione
L'utente si è precedentemente autenticato come titolare del negozio.

Input

Nome titolare – Facoltativo
Cognome titolare – Facoltativo
Città – Facoltativo
Codice Fiscale – Facoltativo
P.I.V.A. – Facoltativo
E-mail – Un solo indirizzo. Facoltativo
Un dato tra quelli elencati deve essere obbligatoriamente inserito per effettuare la ricerca

Elaborazione

Il sistema presenta una lista con l'elenco dei clienti con la possibilità di ricerca.
Selezionando un cliente dalla lista è possibile richiedere la visualizzazione o la stampa della sua scheda riportante tutti i dati anagrafici e contabili del cliente.

Output

La lista con l'elenco dei clienti, visualizzata a schermo o stampata, riporta per ciascuno di essi:
Nome – Cognome – Città – Numero Acquisiti – Data ultimo acquisto – Totale fatturato

Prova adesso!

Descrivi le seguenti funzionalità:
- inserimento di un cliente con verifica e-mail;
- cancellazione di una voce dal carrello;
- elenco degli articoli, con indicazione delle movimentazioni di carico/scarico.

◀ SRS standard IEEE/ANSI 830-1998

Table of contents
1. Introduction
   1.1. Purpose (Scopo del documento)
   1.2. Scope (Scopo del prodotto)
   1.3. Definitions, Acronyms and Abbreviations
   1.4. References
   1.5. Overview (Descrizione generale del resto del documento)
2. General Description
   2.1. Product Perspective
   2.2. Product Functions
   2.3. User Characteristics
   2.4. General Constraints
   2.5. Assumptions and Dependencies
   …
3. Specific Requirements
   3.1. Functional Requirements
   3.1.1. Functional Requirement 1
   3.1.1.1. Introduction
   3.1.1.2. Inputs
   3.1.1.3. Processing
   3.1.1.4. Output
   3.1.2. Functional Requirement 2
   …
   3.1.3.Functional Requirement n
   …
   3.2. External Interface Requirements
   3.2.1. User Interfaces
   3.2.2. Hardware Interfaces
   3.2.3. Software Interfaces
   3.2.4. Communications Interfaces
   3.3. Performances Requirements
   3.4. Design Constraints
   3.4.1 Standard Compliance
   3.4.2. Hardware Limitation
   3.5. Software System Attributes
   3.5.1. Security
   3.5.2. Maintainability
   …
   3.6. Other Requirements
   3.6.1. Logical Database Requirements
   3.6.2. Operations
   3.6.3. Site Adaptation requirements ▶

Il documento completo *IEEE830.pdf* è disponibile nella cartella materiali della sezione dedicata a questo volume del sito http://www.hoepliscuola.it/.

Realizzare un efficace documento SRS

Il documento SRS riporta in modo corretto e non ambiguo le specifiche dei requisiti del software ma non descrive alcun dettaglio progettuale o implementativo e non impone vincoli addizionali al prodotto software, quali per esempio qualità, oggetto di documenti specifici.

Lo standard IEEE 830 descrive alcune caratteristiche fondamentali di cui un efficace SRS deve essere in possesso, a partire dalla validazione dei requisiti.

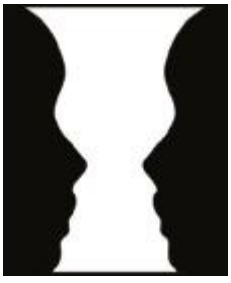
Validazione dei requisiti

Durante il processo di sviluppo i requisiti devono essere continuamente validati da cliente e utenti; la validazione dei requisiti richiede di controllare:

- correttezza: un SRS è corretto se, e solo se, il sistema software soddisfa appieno ciascun requisito in esso riportato; una specifica è corretta se rappresenta accuratamente il sistema che il cliente richiede e che gli sviluppatori intendono sviluppare;
- completezza: una specifica è completa se tutti i possibili scenari per il sistema sono descritti, inclusi i comportamenti eccezionali dovuti a una qualunque combinazione di ingressi (validi o non validi); deve inoltre riportare i riferimenti alle figure, diagrammi e tabelle, la definizione dei termini e delle unità di misure utilizzati;
- consistente: nessun requisito deve essere in conflitto con altri requisiti;
- coerenza: i requisiti non devono contraddirsi tra di loro;
- chiarezza: una specifica è chiara se non è possibile interpretarla in due modi diversi; non possiamo quindi avere ambiguità di descrizione, dovute alla terminologia o semplicemente alla scarsa oppure elevata presenza di dettagli.

ESEMPIO 21 *Ambiguità e dettagli*

Il seguente disegno mostra un caso di ambiguità: se ci si focalizza sui contorni neri si vede il profilo di due volti, se si osserva il bianco si individua un vaso. ▶



Anche il livello di dettaglio può portare all'introduzione di ambiguità: vediamo una versione "arricchita" dello stesso disegno dove è possibile individuare molte più ambiguità: ▶

(da Giorgio Zanetti, zanblog.it).

- realismo: la specifica dei requisiti è realistica se può essere implementata tenendo conto dei vincoli presenti nel dominio del sistema;
- modificabilità: la struttura e lo stile dell'SRS devono essere tali da consentire facili modifiche, preservando consistenza e completezza (un SRS con ridondanze non si modifica facilmente);
- verificabilità: la specifica dei requisiti è verificabile se, e soltanto se, una volta che il sistema è stato costruito, test ripetuti possono essere delineati per dimostrare che il sistema soddisfa i requisiti.
  Un requisito è verificato se esiste un procedimento (di costo compatibile) tramite il quale una persona o una macchina può stabilire se il software soddisfa il requisito.
  Esempi di requisiti non verificabili
  – il programma deve funzionare bene
  – il programma dovrebbe avere una buona interfaccia
  – il tempo di risposta deve essere di norma 10 sec.
  Esempi di requisiti verificabili
  – il tempo di risposta all'evento X deve stare entro 10 sec.
  – il tempo di risposta all'evento Y deve stare entro 10 sec. nel 60% dei casi
- tracciabilità: la definizione IEEE riporta "se l'origine di ciascun requisito è chiara e può essere referenziata nello sviluppo futuro ogni funzione del sistema deve poter essere individuata e ricondotta al corrispondente requisito funzionale".
  È necessario tracciare le dipendenze tra i requisiti e le funzioni del sistema, oltre che per effettuare lo sviluppo di test per poter successivamente introdurre eventuali cambiamenti del sistema.
  La tracciabilità può essere realizzata assegnando un numero univoco a ciascun requisito

<tipo-requisito><numero-requisito>

e collegando i requisiti ai vari aspetti del sistema, una volta che anch'essi sono stati catalogati e numerati, mediante una tabella a doppia entrata come la seguente:

		Aspetti del sistema o del suo ambiente						
		A1	A2	A3	A4	A5	…	An
Requisiti	R1		✔		✔			
	R2		✔	✔				
	…							
	Rm	✔			✔	✔		

TRACCIABILITÀ DI UN SRS

Un SRS è tracciabile se:
- è chiara l'origine di ogni requisito (tracciatura all'indietro, ai documenti precedenti);
- ogni requisito ha un nome o un numero (tracciatura in avanti per i requisiti futuri).

La convalida delle specifiche

La stesura del SRS deve seguire pari passo lo sviluppo del prodotto software dato che generalmente non è possibile specificare tutti i dettagli durante la fase iniziale di raccolta dei requisiti anche perché i requisiti possono cambiare in corso d'opera (sistemi complessi possono avere anche anni come tempi di sviluppo e nel frattempo gli obiettivi/bisogni di una organizzazione possono cambiare).

Il documento dei requisiti dovrebbe essere organizzato in modo tale che le modifiche possano essere eseguite senza la riscrittura del documento.

Anche se la stesura del documento viene realizzata scrupolosamente spesso si commettono errori che, se scoperti in fase avanzata di realizzazione, possono provocare enormi danni (ritardo nello sviluppo e quindi lievitazione dei costi); i tipi di errori più frequenti sono:

- requisiti non chiari: i requisiti sono espressi male o sono state omesse delle informazioni; in questo caso i requisiti devono essere riscritti;
- informazioni mancanti: le informazioni mancano dal documento dei requisiti; l'ingegnere dei requisiti deve raccogliere delle altre informazioni;
- conflitto tra requisiti: sono presenti contraddizioni fra i vari requisiti oppure tra i requisiti e l'ambiente operativo; una negoziazione con il cliente può aiutare a rimuovere il conflitto;
- requisiti non realistici: il cliente dovrebbe essere consultato, i requisiti cancellati, modificati e resi più realistici.

L'individuazione di queste indicazioni viene fatta aiutandosi con una check list che solitamente comprende non più di 10 domande da sottoporre all'utente finale, come quelle riportate nel seguente esempio.

ESEMPIO 22 ***Check list per la convalida dei requisiti***

Esempio di check list

1. Sono state definite tutte le risorse hardware?
2. È stato specificato il tempo di risposta delle funzioni?
3. Sono state specificate tutte le interfacce esterne, hardware, software e relativamente ai dati?
4. Sono state specificate tutte le funzioni richieste dall'utente?
5. È possibile testare ogni requisito?
6. Sono state specificate tutte le risposte a condizioni eccezionali?
7. È stato definito lo stato iniziale del sistema?
8. Sono state specificate le future possibili modifiche?

Non è semplice individuare gli errori presenti in un documento SRS e a tal fine è importante effettuare frequenti revisioni e ispezioni coinvolgendo sempre un rappresentante di tutti gli attori compreso l'autore del documento, il cliente, un progettista, un esperto di qualità: prima di organizzare le riunioni specifiche è necessario consegnare una copia del documento da analizzare a tutti i partecipanti invitandoli a rivedere il documento prima della riunione per poter segnalare in quella sede eventuali osservazioni.

È anche molto utile far leggere il documento a qualcuno diverso dall'autore per coglierne potenziali problemi di ambiguità o incompletezza (*Tecnica di Reading*).

Il primo metodo per eliminare gli errori resta comunque sempre quello "di non farli": concludiamo quindi questa trattazione indicando alcuni semplici consigli per ottenere una specifica precisa e priva di ambiguità:

- evitare termini troppo generici o troppo precisi;
- mantenere un livello di astrazione costante;
- riferirsi allo stesso concetto sempre nello stesso modo così da evitare l'uso di sinonimi (termini diversi con il medesimo significato) e omonimi (termini uguali con differenti significati);
- usare frasi brevi e semplici possibilmente uniformandone la struttura;
- dividere il testo in paragrafi e dedicare ogni paragrafo alla descrizione di una specifica entità della realtà modellata, evidenziandola in ogni paragrafo.

Verifichiamo le competenze

1 Situazione: negozioOnline

In riferimento al sistema *NegozioOnline* descritto nelle precedenti lezioni, descrivi le seguenti funzionalità nel documento SRS secondo lo schema proposto dallo standard IEEE 830:
3.e Registrazione di un nuovo cliente
3.f Cancellazione di un nuovo cliente
3.g Variazione dei dati di un nuovo cliente
3.h Inserimento di un nuovo articolo in magazzino
3.i Eliminazione di un nuovo articolo in magazzino

2 Situazione: biblioteca scolastica

Il sistema dovrà archiviare i dati di una biblioteca scolastica dove oltre ai libri, ai giornali e alle riviste, è presente una sezione multimediale con video, nastri audio e CD-ROM.
Il sistema dovrà permettere agli utenti di fare delle ricerche per titolo, autore, categoria o codice ISBN mentre il bibliotecario deve poter stampare l'elenco dei ritardi di consegna per poter fare i dovuti solleciti.
Il sistema dovrà riconoscere l'utente attraverso la stessa smart-card che lo identifica all'interno della scuola, utilizzata per esempio per rilevare le presenze e i ritardi
Dopo aver individuato e realizzato i diagrammi dei casi d'uso, descrivi le seguenti funzionalità nel documento SRS secondo lo schema proposto dallo standard IEEE 830:
3.m Inserimento di una nuova categoria
3.n Cancellazione di un alunno
3.o Inserimento di un nuovo libro
3.p Elenco dei ritardi di consegna per i libri in base a una data specifica
3.q Elenco dei ritardi di consegna per una classe dell'istituto

3 Situazione: gite scolastiche

Il sistema richiesto serve per gestire le gite scolastiche e deve essere in grado di memorizzare nuovi itinerari con le seguenti informazioni: destinazione, giorni, tipo, descrizione, numero minimo e massimo di partecipanti, costo, anno di corso, optional.
Possono essere presenti più escursioni/itinerari per la medesima destinazione
I tipi si distinguono in base alla regione e/o alla nazione di destinazione.
Gli optional riguardano il vitto, se è compreso, escluso, solo cena ecc.
Il sistema deve permettere di prenotare un itinerario da parte di una o più classi, in base al numero richiesto di partecipanti: all'atto della prenotazione devono essere raccolte le autorizzazioni da parte dei genitori per gli alunni minorenni e un acconto pari al 10% dell'importo.
Il sistema deve permettere di modificare/cancellare un itinerario solo se non sono state fatte prenotazioni: deve anche permettere di annullare una escursione prenotata in caso di motivazioni gravi (eventi atmosferici, bellici, motivi didattici ecc.).
Un singolo alunno può rinunciare a una gita portando la documentazione medica: in questo caso il costo totale a saldo deve essere ripartito sul resto della comitiva.

4 Dopo aver individuato e realizzato i diagrammi dei casi d'uso, descrivi le seguenti funzionalità nel documento SRS secondo lo schema proposto dallo standard IEEE 830:

3.k Inserimento di un nuovo itinerario per una vecchia destinazione
3.l Inserimento di un nuovo itinerario per una nuova destinazione
3.m Cancellazione di un alunno da una gita
3.n Stampa del programma di un particolare itinerario scelto in base alla destinazione
3.n Elenco delle escursioni per tutte le classi di un anno di corso
3.n Elenco delle escursioni effettuate da una classe nei diversi anni

Esercitazioni di Laboratorio 1

LA REALIZZAZIONE DEGLI SCHEMI UML CON STARTUML

Generalità

Esistono molti prodotti che permettono di realizzare gli schemi UML, tra i quali ricordiamo:

- StarUML, scaricabile gratuitamente all'indirizzo http://staruml.sourceforge.net/en oppure dalla cartella materiali della sezione dedicata a questo volume del sito http://www.hoepliscuola.it/;
- Visual Paradigm for UML Community Edition, scaricabile gratuitamente all'indirizzo http://resource.visual-paradigm.com/vpsuite3.0sp1/support_uml2_xmi.html;
- ArgoUML, pacchetto open source di modellizzazione dei diagrammi con lo standard UML 1.4 diagrams scaricabile gratuitamente all'indirizzo http://argouml.tigris.org;
- Microsoft Visio, strumento a pagamento per la creazione di diagrammi che offre anche la possibilità di condivisione dei progetti sul Web in tempo reale.

In questa esercitazione presenteremo StarUML, un progetto open source per lo sviluppo rapido, flessibile, estensibile di diagrammi UML, secondo il paradigma MDA (Model Driven Architecture).

Zoom su...

MODEL DRIVEN ARCHITECTURE

La metodologia Model Driven Architecture (MDA) fornisce un insieme di linee guida (standard) intese a permettere un approccio integrato allo sviluppo del software, dove i modelli sono considerati parte integrante del processo di implementazione. Il suo utilizzo generalmente ha numerosi vantaggi in termini sia di flessibilità per l'implementazione, l'integrazione e la manutenzione, il collaudo e la simulazione, sia per quanto riguarda la portabilità, la interoperabilità e riusabilità in un ampio arco di tempo.

StarUML è ricco di funzioni, flessibile ed estendibile grazie alla sua architettura a plug-in e alla disponibilità di apposite API; con una semplice interfaccia IDE consente la composizione dei diagrammi UML di base: Use Case, Class, Sequence, Collaboration, Statechart, Activity, Component e Deployment.

Attualmente StarUML è aggiornato all'UML 2.0 dato che segue continuamente le modifiche che l'OMG (◀ Object Management Group ▶) apporta alle specifiche UML.

◀ **Object Management Group OMG's** mission is to develop, with our worldwide membership, enterprise integration standards that provide real-world value. OMG is also dedicated to bringing together end-users, government agencies, universities and research institutions in our communities of practice to share experiences in transitioning to new management and technology approaches like Cloud Computing. ▶

È scritto in Delphi ma con l'installazione degli appositi plug-in, StarUML fornisce pieno supporto ai linguaggi più diffusi quali C/C++, Java, Visual Basic, Delphi, JScript, VBScript, C#, VB.NET e offre funzionalità uniche come quelle che consentono la creazione automatica di documenti della suite Microsoft Office (Word, Excel, PowerPoint).

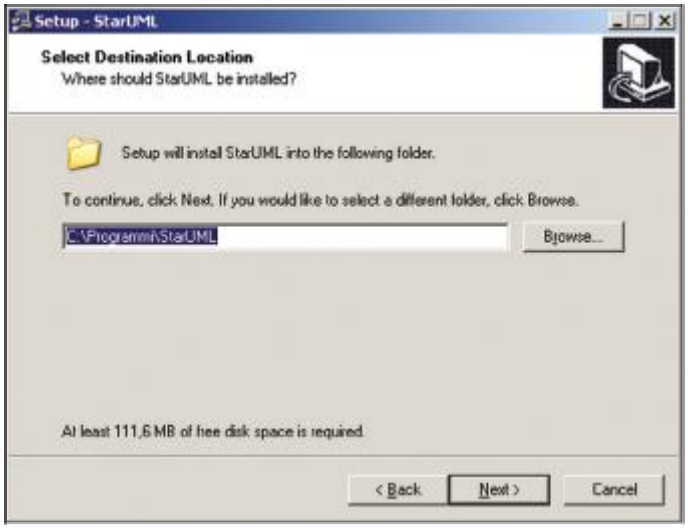
È apprezzata particolarmente la possibilità di effettuare la generazione di codice a partire dai diagrammi.

Installare StartUML

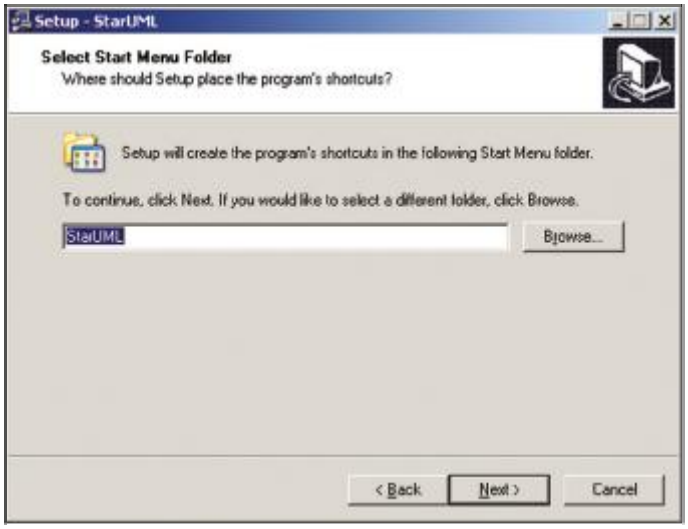
Dopo aver scaricato il staruml-5.0-with-cm.exe, mandiamolo in esecuzione con un doppio click, e la prima videata che ci viene presentata è la classica finestra di benvenuto: ▶

Si procede cliccando su Next che ci porta alla successiva videata dove viene richiesta l'accettazione della Licenza d'uso. ▶

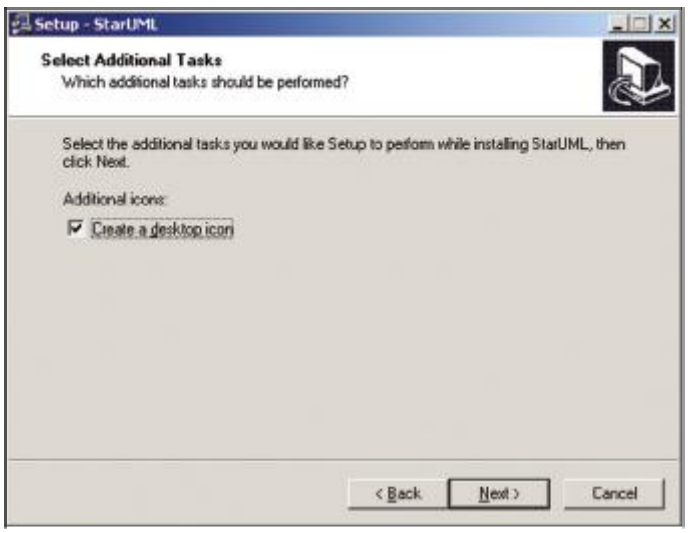
Dopo aver confermato l'accettazione si attiva il pulsante Next e si procede con la scelta della directory di installazione: ▶



Procediamo senza modificare quella proposta in automatico e confermiamo anche la successiva richiesta di creazione di un shortcut nel menu principale: ▶

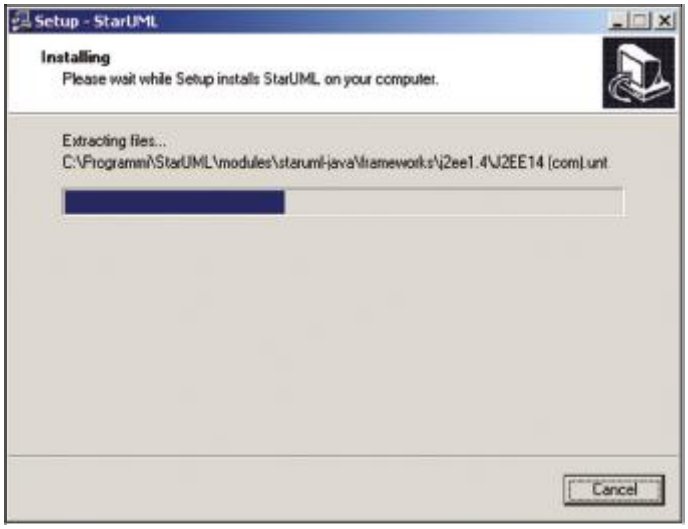


e quella di creare una icona sul desktop: ▶



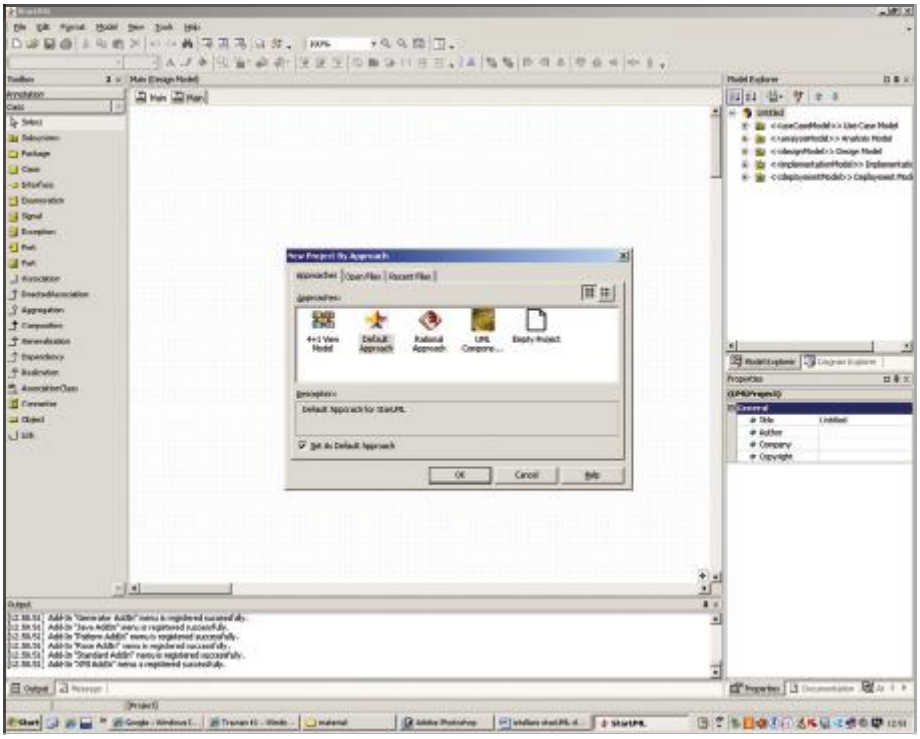
Viene ora visualizzata una videata che riassume le scelte fatte sino a ora per confermare l'inizio dell'installazione del programma sul nostro disco fisso: ▶

Cliccando su Install si avvia l'estrazione e la copia dei file del programma: ▶

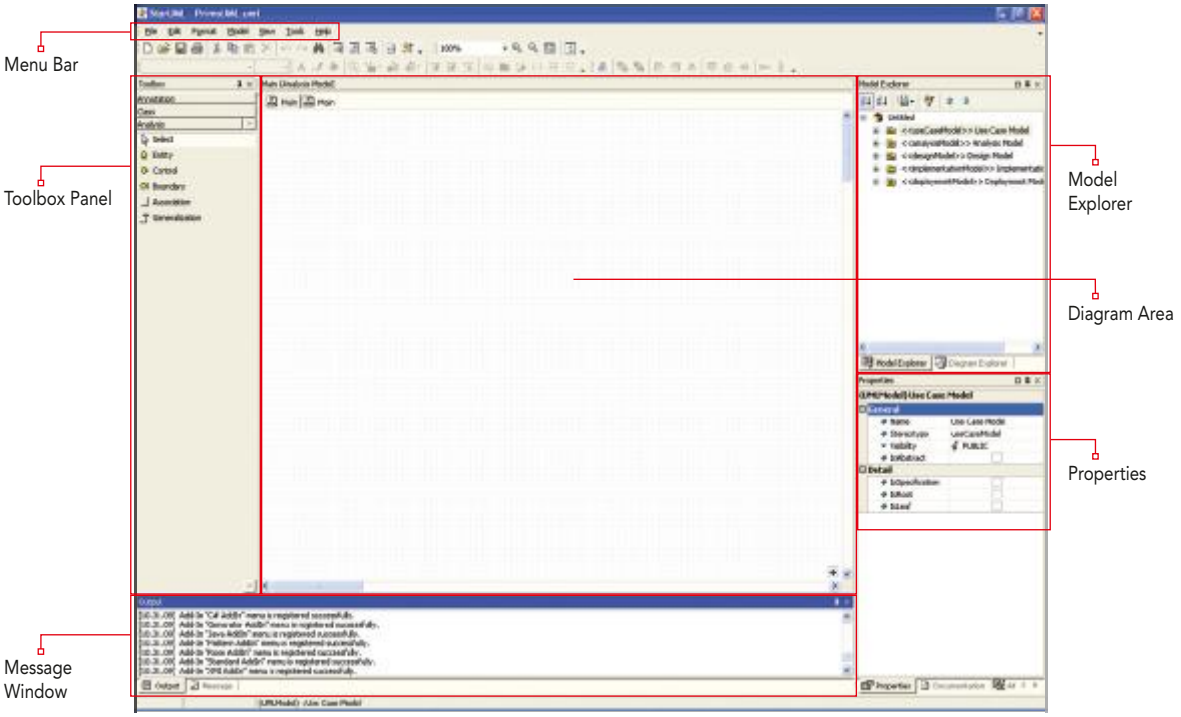


Se tutto va a buon fine, viene visualizzata la seguente videata: ▶

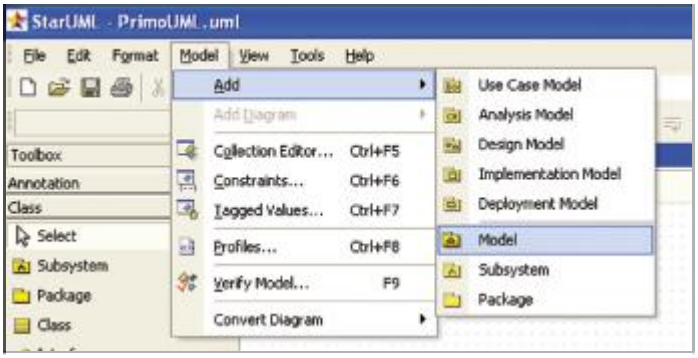
Avviamo il programma StarUML e selezioniamo Default Approach tra le cinque opzioni che ci vengono proposte:



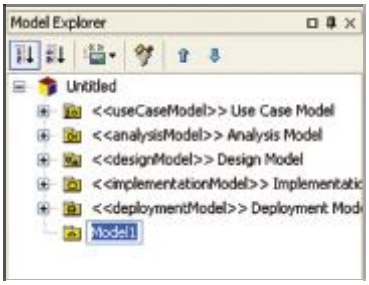
Osservando la schermata principale individuiamo le seguenti sezioni:



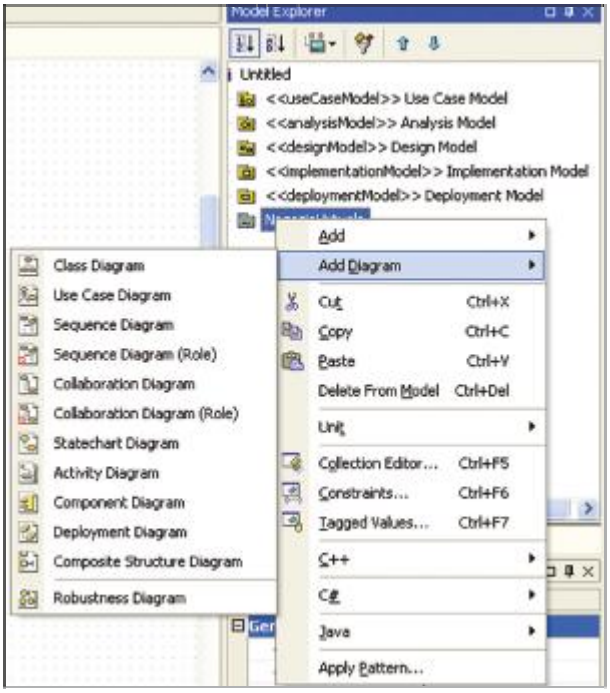
Aggiungiamo un nuovo modello a quelli proposti automaticamente:



che verrà visualizzato nella sezione Model Explorer (dove potremo modificarne il nome a nostro piacere, per esempio NegozioVirtuale):

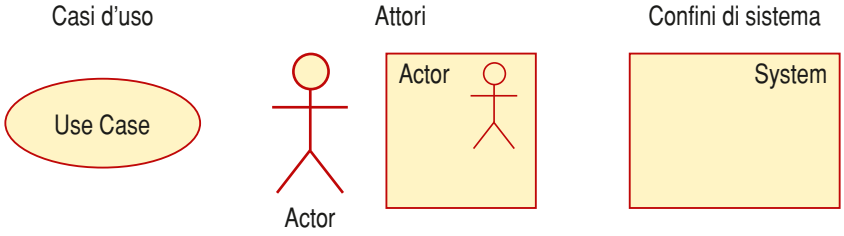


Aggiungiamo a questo modello un diagramma direttamente dal menu che viene presentato cliccando col tasto destro sul modello appena creato, per esempio un Use Case Diagram:

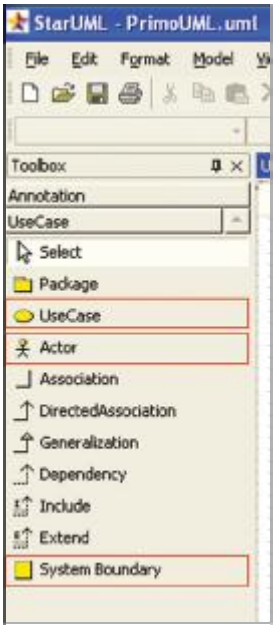


Nella barra del Diagram Area ci viene aperta una nuova finestra: ▶

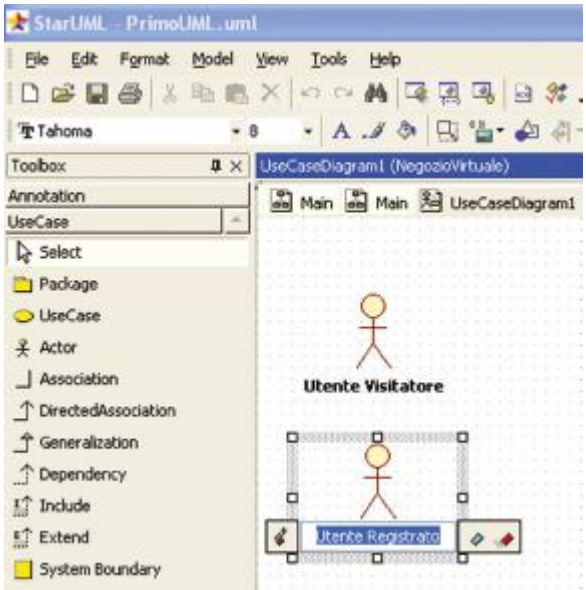
Realizziamo per esempio il diagramma dei casi d'uso dell'esempio NegozioVirtuale descritto nella lezione 4 utilizzando i seguenti simboli grafici di StarUML:



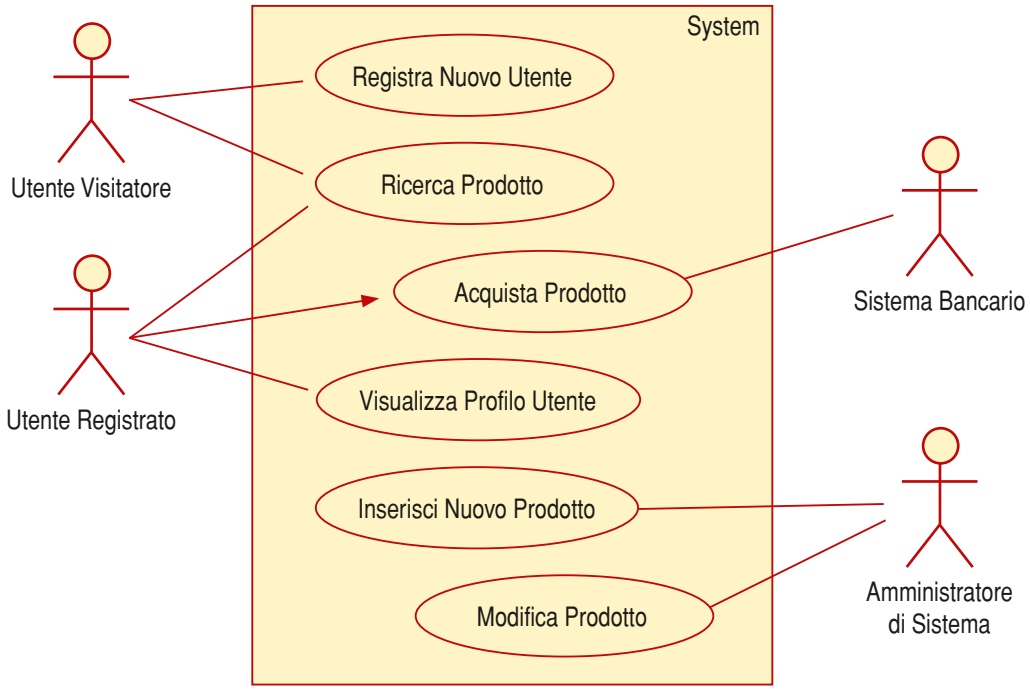
che troviamo nel Toolbox Panel non appena selezioniamo sulla barra UseCaseDiagram1 ▶



Inseriamo quindi gli elementi nell'area di lavoro a partire dagli attori, selezionando l'icona sul Toolbox Panel e successivamente cliccando sull'area di lavoro nella posizione in cui si desiderano collocare:



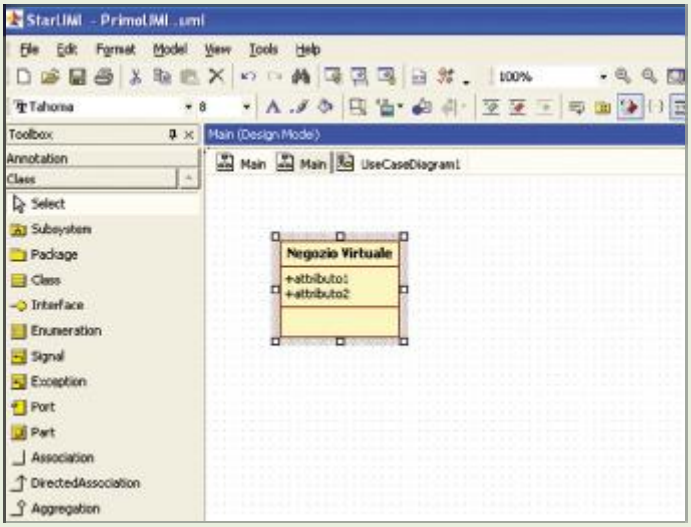
Assegnando a ciascuno il proprio nome; analogamente si inseriscono gli Use Case ottenendo il seguente diagramma



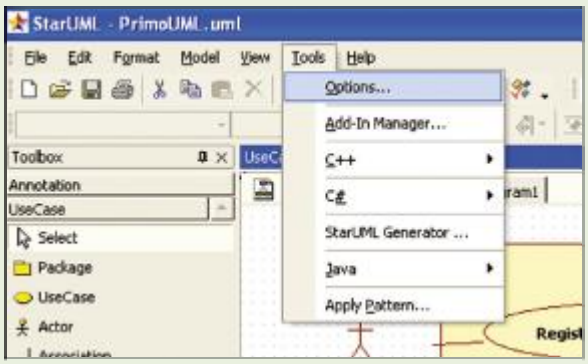
Zoom su...

GENERAZIONE CODICE SORGENTE

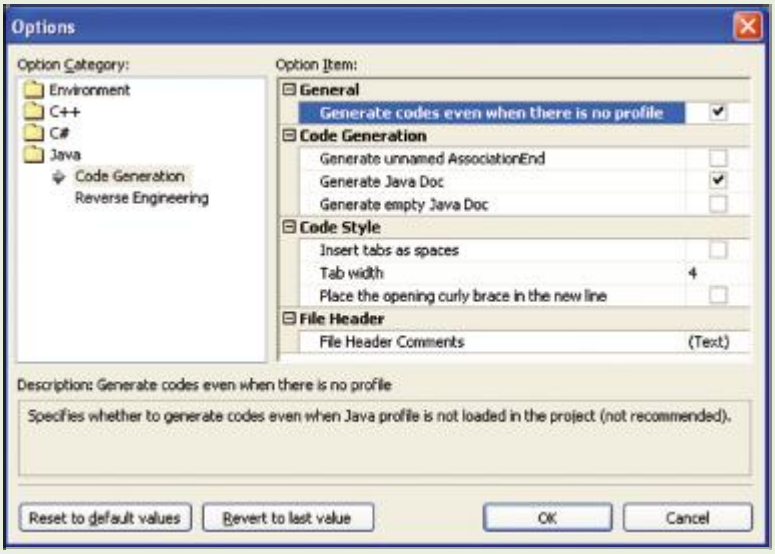
StarUML permette di generare automaticamente il codice sorgente in Java, C++ oppure in C#. È necessario dapprima creare almeno una classe nella sezione Design Model, come indicato nella seguente figura:



Per poter generare il codice Java è opportuno effettuare una operazione preliminare: selezionando ora Option dal menu Tools: ▶



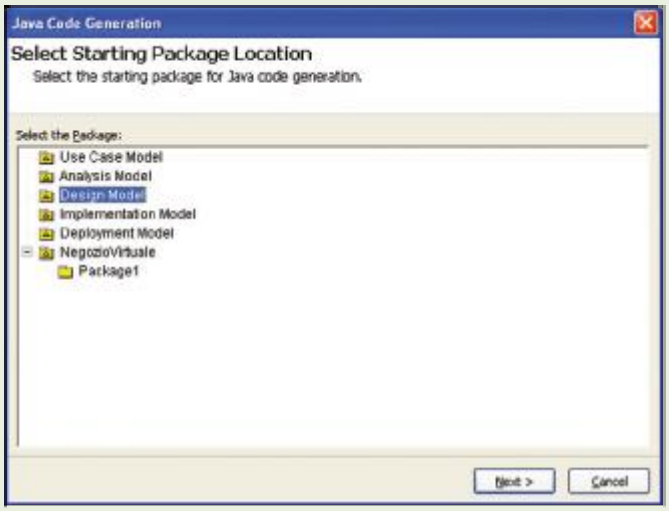
nella cartella Java Code Generation è necessario spuntare la voce Generate codes even where there is no profile nella sezione general, come riportato nella seguente figura:



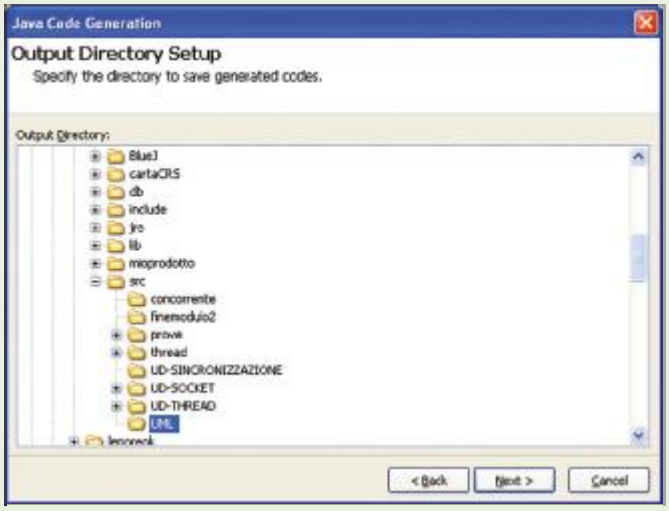
È anche necessario creare anticipatamente la cartella dove il programma andrà a memorizzare i file sorgenti Java che verranno creati; nel nostro caso abbiamo definito la cartella c:\java7\src\UML.

Per procedere con la generazione del codice è possibile selezionare direttamente la classe col tasto destro e scegliere l'opzione dal menu che ci viene proposto oppure scegliere Java dal menu Tools della barra del menu principale:

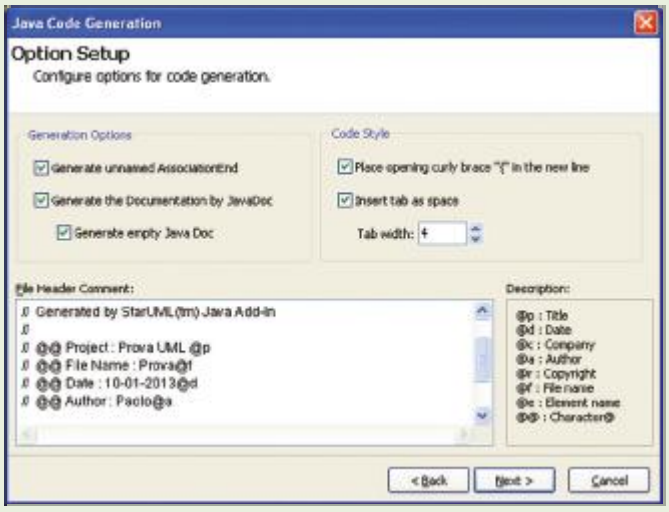
Seguendo le indicazioni proposte dalle successive videate ▶



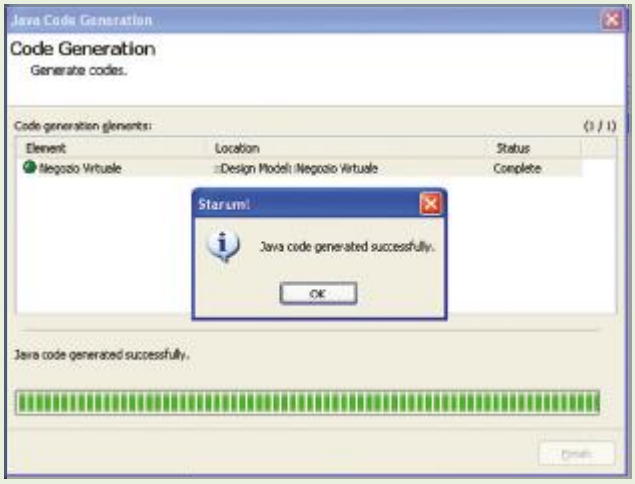
selezioniamo per i file .java la directory di destinazione prima creata ▶



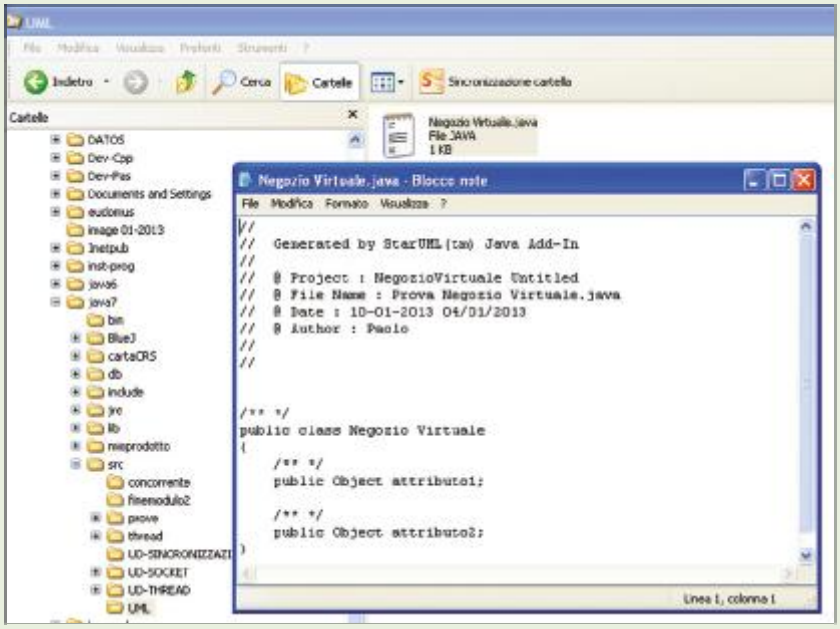
Ci viene proposta la seguente videata nella quale possiamo aggiungere i commenti JavaDoc al nostro codice sorgente: ▶



Sempre procedendo con Next > viene generato il codice sorgente e ci viene proposta l'ultima finestra dove sono elencati i file generati (in questo caso solo la classe NegozioVirtuale):



Posizioniamoci sul nostro disco fisso e nella cartella specificata è ora presente un file .Java che visualizziamo direttamente con un editor di testo:



Il pacchetto StarUML offre molteplici possibilità: si consiglia di approfondirne la conoscenza consultando la documentazione on-line presente ai seguenti indirizzi:

- *StarUML5.0 User Guide*: http://staruml.sourceforge.net/docs/user-guide(en)/toc.html
- *StarUMLTutorial*: http://cnx.org/content/m15092/latest/
- *StarUML_5.0_Developer_Guide*: http://staruml.sourceforge.net/en/documentations.php (la guida StarUML_5.0_Developer_Guide.pdf è anche disponibile nella cartella materiali della sezione dedicata a questo volume del sito http://www.hoepliscuola.it/).

Prova adesso!

Disegna il diagramma dei casi d'uso nelle seguente situazioni.

1 Sistema biblioteca

In una biblioteca sono stati individuati i seguenti requisiti per il bibliotecario:

- solo il bibliotecario può effettuare il prestito dei libri;
- il bibliotecario riceve i libri restituiti dagli utenti;
- effettua la gestione dei libri;
- classifica i libri nuovi, ripone i libri sugli scaffali, segnala i libri danneggiati, ordina nuovi libri.

2 Sistema ordini

In un sistema che raccoglie gli ordini di acquisto sono stati individuati i seguenti requisiti:

- i clienti registrati effettuano gli ordini;
- i clienti effettuano il pagamento comunicando i dati dell'acquisto al sistema che procede a processare l'ordine;
- se la merce è disponibile, l'ordine viene evaso, altrimenti si colloca in attesa;
- quando un fornitore consegna della merce si verifica se sono presenti ordini in attesa per procedere con l'evasione;
- si individua quali ordini possono essere evasi;
- si effettuano le assegnazioni della nuova merce agli ordini in attesa e la merce rimanente viene sistemata in magazzino.

3 Negozio di Musica

Un negozio di musica vende anche libri e riviste musicali.
Si intende automatizzare l'intero processo, dall'approvvigionamento alla vendita, a partire dalla gestione del magazzino, dove vengono richieste le seguenti funzionalità:

- la verifica della disponibilità di un prodotto (caso d'uso *VerificaDispProdotto*);
- l'effettuazione di un ordine al fornitore qualora la disponibilità non sia sufficiente (caso d'uso *EffettuaOrdineFornitore*), che richiede la seguente sequenza di attività:
- creazione di un ordine al fornitore (inizialmente vuoto);
- aggiunta del/dei prodotto/i da ordinare;
- invio dell'ordine al fornitore.

4 Sistema bancario

Nella realizzazione di un sistema bancario sono stati individuati i requisiti di due funzionalità di seguito descritte; disegna il digramma dei casi d'uso.

4.1) Apertura conto corrente

Scenario base:

1 il cliente si presenta in banca per aprire un nuovo c/c
2 l'addetto riceve il cliente e fornisce spiegazioni
3 il cliente accetta le condizioni e fornisce i propri dati
4 l'addetto verifica se il cliente è censito in anagrafica
5 l'addetto crea il nuovo conto corrente
6 l'addetto segnala il numero di conto al cliente

Varianti:

3a) se il cliente non accetta le condizioni il caso d'uso termina

3b) se il conto va intestato a più persone vanno forniti i dati di tutti i cointestatari

4a) se il cliente (o uno dei diversi intestatari) non è censito l'addetto provvede a registrarlo, richiede al cliente la firma dello specimen e ne effettua la memorizzazione via scanner

4.2) Inserimento conto corrente

Scenario base:

1 l'addetto richiede al sistema la transazione di inserimento nuovo conto
2 il sistema richiede i codici degli intestatari
3 l'addetto fornisce i codici al sistema
4 il sistema fornisce le anagrafiche corrispondenti e richiede le condizioni da applicare al conto
5 l'addetto specifica le condizioni e chiede l'inserimento
6 il sistema stampa il contratto con il numero assegnato al conto

Varianti:

3a) se il sistema non riconosce il cliente o se fornisce un'anagrafica imprevista, l'addetto può effettuare correzioni o terminare l'inserimento

4 DOCUMENTAZIONE DEL SOFTWARE

UNITÀ DI APPRENDIMENTO

L 1 La documentazione del progetto

L 2 La documentazione del codice

OBIETTIVI

- Comprendere la necessità di documentare
- Sapere quali sono i documenti necessari in un progetto
- Conoscere il concetto di documentazione interna ed esterna
- Apprendere le modalità per realizzare la documentazione esterna di sistema e utente
- Acquisire una tecnica di documentazione del codice
- Saper definire uno standard di documentazione
- Conoscere i principali tool di documentazione automatica del codice

ATTIVITÀ

- Saper effettuare la documentazione del codice
- Utilizzare Javadoc come strumento di documentazione automatica
- Installare e utilizzare Doxygen come strumento di documentazione automatica
- Installare e configurare Subversion e TortoiseSVN
- Utilizzare TortoiseSVN per effettuare il controllo delle versioni

LEZIONE 1

LA DOCUMENTAZIONE DEL PROGETTO

IN QUESTA LEZIONE IMPAREREMO...

- la necessità di documentare
- la documentazione esterna di sistema
- la documentazione esterna utente
- i tool di documentazione del codice

Generalità

La documentazione di un progetto software deve seguire, per quanto possibile, le fasi definite nel processo di sviluppo del sistema e deve prevedere la stesura di un insieme di documenti in relazione temporale con il completamento delle fasi del processo di sviluppo.

Lo sforzo dedicato alla documentazione del software non deve essere visto come un'attività costosa e noiosa, alla quale associare la minima priorità, poiché i benefici di una buona documentazione sono sempre tangibili: non a caso, molti fattori di qualità sono direttamente o indirettamente influenzati in maniera positiva dalla presenza di documentazione.

La documentazione, man mano che viene prodotta, diviene un supporto al processo di sviluppo del software e quindi deve essere redatta durante lo svolgimento del progetto stesso e non al suo termine solo al fine di completare il "corredo" del pacchetto software.

Non esiste uno standard che descrive rigorosamente i documenti che devono essere prodotti: in questa lezione si è cercato di riunire tutta la serie di elaborati cartacei che *dovrebbero* essere prodotti durante lo sviluppo di un prodotto software di medie dimensioni, come indicato dai maggiori teorici dell'ingegneria del software.

Standard della documentazione

È importante definire un proprio standard nella produzione della documentazione da mantenere per tutti i documenti che vengono prodotti; qui elenchiamo un insieme di punti che devono essere definiti dal responsabile del progetto.

Ⓐ Standard per la produzione di un documento:
- per i documenti: descrivono la struttura, il contenuto e l'editing del documento;
- per la presentazione di un documento: definiscono i font, stili, l'uso di un logo....

ESEMPIO 1

Tutti i documenti dovranno contenere:
- il nome e il logo della società;
- l'elenco dei nominativi dei redattori, con le rispettive firme e la data;
- l'elenco dei nominativi di chi ha approvato il documento con le rispettive firme e la data;
- l'oggetto del contenuto ed eventualmente un sommario.

Inoltre tutte le pagine dei documenti devono essere numerate progressivamente, indicando sempre anche il totale delle pagine (esempio pag. 4 di 10).

Ⓑ Standard per la manutenzione di un documento:
- per l'identificazione di un documento: come i documenti sono codificati per essere univocamente identificati;
- per l'aggiunta di un documento: come i documenti vanno codificati e validati; come gestire le versioni con il registro delle funzionalità apportate con i vari riferimenti, il numero e la data;
- per l'aggiornamento di documenti: definire come le modifiche di una precedente versione si riflettono in un documento, con i vari riferimenti, il numero e la data.

Ⓒ Standard per la distribuzione di un documento:
- standard per lo scambio/condivisione di documenti: come i documenti vanno memorizzati e scambiati/condivisi tra differenti sistemi di documentazione.

ESEMPIO 2

Per esempio, si può stabilire che per poter permettere lo scambio di documenti elettronici prodotti usando differenti sistemi e computer, sia in modo diretto che per l'inoltro per mezzo di e-mail, il loro sia lo standard XML (questa motivazione viene anche dal fatto che i documenti devono essere archiviati e la vita di un sistema di word processing potrebbe essere più breve di quella del software che si sta documentando).

Nella documentazione un aspetto fondamentale è quello di permettere la *tracciabilità* delle modifiche apportate al software o alla architettura del sistema: se è agevole risalire a tutta la "cronologia" che ha portato a un modifica o alla realizzazione di codice si riduce notevolmente lo sforzo necessario per comprenderlo.

Documentazione del progetto

La bontà di un progetto viene valutata anche in base alla documentazione acclusa: occorre pertanto che la documentazione sia redatta in forma standard, completa e comprensibile e, a parità di contenuto e di chiarezza, è preferibile che la documentazione sia il più possibile concisa perché un testo lungo non invoglia la lettura.

La documentazione del progetto si compone di tre parti:
- il manuale per l'utente;
- la documentazione tecnica (che comprende anche il codice sorgente);
- le prove di collaudo.

Possiamo classificare la documentazione di un progetto secondo quattro modalità:

Ⓐ esterna o interna:
- **esterna:** separata dal programma vero e proprio;
- **interna**: sotto forma di commenti o help contenuti nel file sorgente;

Ⓑ in linea o fuori linea:
- **in linea**: richiamabile su richiesta durante l'esecuzione del programma,
- **fuori linea**: sotto forma di manuale consultabile separatamente;

Ⓒ globale o locale:
- **globale:** riguardante il programma nel suo complesso;
- **locale:** sotto forma di commenti inseriti nel codice sorgente in punti specifici del programma;

Ⓓ per l'utente o per il programmatore:
- **rivolta all'utente:** per chi usa il programma, generalmente organizzata in:
  – manuale d'uso:
  – manuale di installazione;
  – help in linea;
- **rivolta alla programmazione**: è specifica per il personale tecnico interessato alla struttura del programma per lo sviluppo o per la sua manutenzione, e comprende
  – documentazione inerente il progetto e il management del progetto;
  – documentazione del codice.

La documentazione di un sistema quindi non riguarda soltanto il codice sorgente che costituisce il programma vero e proprio ma anche:
- le specifiche funzionali e non funzionali del sistema;
- le scelte di progetto riguardanti l'architettura generale, le strutture degli archivi e dei dati in memoria centrale e i principali algoritmi utilizzati;
- le modalità d'uso del programma;
- i collaudi effettuati e i loro risultati.

In questa lezione descriviamo la documentazione esterna, mentre la documentazione interna sarà trattata nella prossima lezione.

La documentazione esterna

Dopo l'approvazione del preventivo si avvia la realizzazione di un sistema software e viene nominato un direttore generale responsabile del progetto (project manager).
La prima attività del project manager è proprio la stesura dell'elenco dei documenti necessari allo svolgimento del progetto che devono tenere traccia scritta di tutte le attività inerenti al progetto stesso.
Questa documentazione deve consentire alla direzione di pianificare e controllare lo svolgimento del progetto ma anche di analizzare a posteriori ciò che è accaduto, in un'ottica di miglioramento sia dell'organizzazione che del processo di produzione del software.

Possiamo individuare due gruppi di documenti:

1 documentazione inerente il management del progetto:
- organigramma
- diario di progetto
- verbale
- piano di progetto
- norme di progetto

- offerta
- contratto per lo sviluppo, la fornitura del software e l'assistenza all'avviamento
- piano di gestione della qualità
- relazione finale

2 documentazione del progetto:
- analisi del dominio
- analisi dei requisiti (documento SRS)
- specifica architetturale
- specifica di dettaglio
- piano delle prove
- risultati delle prove
- manuale d'uso

Ogni documento redatto dal gruppo di progetto deve comparire nel diario di progetto che il **project manager** mantiene costantemente aggiornato anche perché le eventuali verifiche ispettive terranno conto solo ed esclusivamente dei documenti elencati nel diario di progetto.

Descriviamoli succintamente uno per uno.

Documentazione inerente il management del progetto

Organigramma

È un documento che contiene l'elenco dei membri del gruppo di progetto con la descrizione dei ruoli assegnati a ciascuno di essi e viene redatto dal project manager prima dell'inizio delle attività. Ogni ruolo viene descritto indicando le competenze di ciascun membro del progetto ed è soggetto a modifiche dato che nel corso dell'evoluzione del progetto l'assegnazione di qualche ruolo potrebbe essere rivista.
Viene sottoscritto da tutti membri del team in modo che ciascuno conosca i propri incarichi.
Non è soggetto a numerazione di versioni in quanto è a uso solo del project manager: in casi di variazione dell'organigramma può essere sostituito con un nuovo documento.

Diario di progetto

Rappresenta il **registro ufficiale** della documentazione del progetto e contiene l'elenco di tutti i movimenti relativi all'archivio dei documenti del progetto e del software che si sta sviluppando.
Ogni movimento viene registrato indicando la data in cui avviene, una descrizione comprendente il suo riferimento univoco (codifica) e la versione del documento (o del prodotto), il nominativo e il ruolo del consegnatario o del ricevente.

Tutti i documenti entranti nell'archivio devono essere approvati dal project manager.

Verbale

Tutte le riunioni del gruppo devono essere verbalizzate, sia che vengano fatte solo dal gruppo di sviluppo sia che prevedano la presenza del committente e/o degli stakeholder.
Ogni riunione viene convocata specificando precedentemente l'ordine del giorno e comunicandolo a tutte le persone che devono parteciparvi: lo svolgimento di una riunione segue tale ordine e nel verbale, dopo l'indicazione della data, del luogo in cui si è tenuta la riunione e dell'elenco delle persone presenti, si riporta sinteticamente quando emerso dalla discussione indicando nominalmente gli interventi in modo da poterli tracciare.

Al termine della riunione viene redatto un verbale che riporti il nome del compilatore e che, dopo una rilettura, viene fatto firmare da tutti i presenti.

È buona norma aggiungere come ultimo punto dell'ordine del giorno la voce "*Varie ed eventuali*" in modo da poter affrontare anche la discussione di esigenze sopraggiunte successivamente alla convocazione della riunione.

Piano di progetto

Il *project manager* deve compilare e tenere aggiornato il piano del progetto che contiene le attività pianificate, in particolare:

1. la definizione degli obiettivi;
2. l'analisi dei rischi;
   - piano gestione rischi;
   - sintesi rischi individuati;
   - strategie di prevenzione;
3. descrizione del modello di processo di sviluppo e delle singole fasi (◀ milestones ▶);
4. stima dei costi;
5. attività di progetto (Work Breakdown Structure, Diagramma di Gantt);
6. consuntivo attività;
7. strumenti utilizzati.

◀ **Milestone** A milestone is a scheduled event signifying the completion of a major deliverable or a set of related deliverables. A milestone has zero duration and no effort – there is no work associated with a milestone. It is a flag in the workplan to signify some other work has completed. ▶

Il piano di progetto si divide in due parti, *pianificazione*, che serve per stimare realisticamente le risorse, i costi e i tempi necessari alla realizzazione del progetto e *consuntivazione*, per confrontare e avere un riscontro tangibile tra le attività effettuate e quelle previste, individuare i punti di discordanza e rendere possibile la pianificazione delle attività future.

Il piano di progetto è corredato da un documento che tiene conto, nel corso del tempo, delle variazioni che sono state apportate, il *registro delle modifiche*, indicando la causa, la data e un numero progressivo, che indica la versione del piano e lo identifica univocamente.

Norme di progetto

Il *project manager* ha anche il compito di redarre un documento che contiene le norme che devono essere rispettate durante le svolgimento del progetto; è generalmente strutturato in due parti:

1. Convenzioni generali
   - documentazione del progetto: convenzioni utilizzate per stilarla, per identificarla, per archiviarla, diffonderla, approvarne i contenuti e integrarli.
   - comunicazione: come viene organizzata la comunicazione tra i membri del gruppo di sviluppo e verso l'esterno, le regole di utilizzo della posta elettronica, le modalità di comunicazione e di gestione dei problemi.
   - organizzazione dello spazio di lavoro: come organizzare i file di progetto, dove archiviarli, come viene strutturata la directory e le varie cartelle, la modalità di accesso e condivisione, l'indirizzo IP o URL del server che conterrà i codici sorgenti e la documentazione, la modalità di aggiornamento e integrazione dei file e dei documenti ecc.

- uso degli strumenti: elenco degli strumenti utilizzati, delle piattaforme di sviluppo, dei compilatori, delle convenzioni usate per l'analisi, la progettazione, la codifica e la verifica del prodotto, e inoltre per la gestione delle versioni.

2. Norme di sviluppo
   - norme di analisi: norme che riguardano le attività inerenti lo svolgimento delle attività di analisi;
   - norme di progettazione: modalità per lo svolgimento delle attività di progettazione e descrizione delle convenzioni adottate per la stesura della documentazione (per esempio, il formato dei diagrammi UML, le modalità di specifica dei singoli componenti ecc).
   - norme di codifica: sono previste singole sezioni dedicate ai diversi linguaggi di programmazione usati nel progetto, dove elencare le convenzioni e gli standard di codifica (regole per l'indentazione, il naming delle variabili, l'intestazione dei file, le modalità di utilizzo degli strumenti di documentazione automatica utilizzati: per esempio *Javadoc* o *Doxygen* descritti in seguito).

Offerta

L'offerta (o preventivo) è il documento che viene sottoposto al cliente prima dell'inizio dello sviluppo del sistema e deve essere firmato dallo stesso per accettazione: viene stilata generalmente dal **project manager** e da un commerciale e comprende una breve descrizione del prodotto offerto, i dettagli economici, l'analisi dei requisiti e i tempi di consegna con l'indicazione delle eventuali penali.

Contratto per lo sviluppo, la fornitura del software e l'assistenza all'avviamento

Al momento della firma per accettazione dell'offerta viene redatto il contratto per lo sviluppo, la fornitura del software e l'assistenza all'avviamento che, oltre alle Condizioni Generali e agli elementi essenziali del contratto, riporta le clausole accidentali raccomandate:

A. limiti delle responsabilità della software house (oltre a quanto previsto dalla legge);
B. modalità di utilizzo corretto del software prodotto;
C. proprietà dei materiali (software) e degli elaborati;
D. tipo di fornitura del codice (sorgente e/o oggetto);
E. criteri, modalità, obiettivi del collaudo;
F. termini di consegna;
G. eventuali penali;
H. periodo di prova del software a decorrere dalla data di collaudo;
I. tipo di assistenza nel periodo di avviamento;
L. attività del cliente che costituiranno parte delle obbligazioni contrattuali del cliente:
   1. indicazione del responsabile del progetto che è referente per il cliente;
   2. definizione completa dei requisiti di progetto (specifiche di sistema, specifiche software);
   3. partecipazione al collaudo e approvazione;
   4. attività di installazione e configurazione sistema (ambiente di produzione e ambiente di test) se non facenti parte esplicitamente delle attività di progetto a carico del professionista.

Piano di gestione della qualità

A seconda delle diverse certificazioni di qualità possedute dalla azienda sviluppatrice deve essere definito il piano della qualità comprendente le politiche e gli obiettivi prefissi, in esso devono essere indicati gli strumenti e le procedure di controllo indicando tempi, tecniche, metodi (Management reviews, Technical reviews, Inspections, Audits), azioni da intraprendere in caso di non conformità e anomalie.

Relazione finale

Alla conclusione dello sviluppo e del collaudo del sistema viene redatta dal project manager la relazione finale, che contiene la descrizione delle funzionalità implementate, dell'architettura del sof-

tware e dell'hardware dove è stato installato e collaudato il sistema, le informazioni sui linguaggi di programmazione e sull'ambiente di sviluppo impiegati, la descrizione delle strutture dei dati realizzate, dell'interfaccia software per l'utente e delle risorse utilizzate dal progetto, la documentazione del collaudo fatto in presenza del committente e da lui sottoscritto come approvazione della fornitura.

Alla relazione finale viene generalmente allegato il manuale operativo per il corretto utilizzo del programma e dei dati da esso generati e viene proposto al cliente un contratto di manutenzione del software per specificare gli accordi economici sullo sviluppo successivo per le migliorie, per lo sviluppo di funzionalità per esigenze tecnico/organizzative al momento non previste e gli eventuali adeguamenti legislativi e un contratto di assistenza, che può essere per esempio costituito da un "pacchetto "di giornate di assistenza oltre che alla assistenza telefonica o fatta mediante accesso remoto.

Documentazione del progetto

Analisi del dominio

Nell'analisi del dominio sono presenti i dati necessari a conoscere l'ambito di sviluppo del prodotto, una breve descrizione della azienda e del progetto da realizzare, la descrizione dei compiti e delle procedure e una analisi dettagliata del dominio, a partire dalla definizione del glossario contenente le definizioni, gli acronimi e le abbreviazioni d'uso comune nel settore d'intervento, le caratteristiche dei clienti e degli utenti, la descrizione della concorrenza e dei prodotti software competitori da loro utilizzati.

Analisi dei requisiti

Il documento di Specifica dei Requisiti Software (SRS) è stato descritto nella lezione 4 dell'unità di apprendimento 3 e descrive quello che è richiesto allo sviluppatore del sistema come risultato del lavoro di analisi tra il cliente, l'utente e lo sviluppatore.

Specifica architetturale

In questo documento viene descritta l'architettura del sistema, la sua decomposizione in moduli funzionali con la definizione delle informazioni che vengono scambiate tra di essi e con l'ambiente esterno, la descrizione dell'articolazione del programma in sotto programmi con particolare riguardo all'albero di chiamata dei sotto programmi stessi.
Nel caso di programmazione a oggetti vengono descritte le singole classi, le gerarchie, la modellizzazione dinamica e comportamentale dei singoli componenti.
Vengono inoltre descritte le interfacce utente e i dati presenti in ciascuna di esse.

Specifica di dettaglio

Nella specifica di dettaglio si riprendono i componenti software (modulo, funzioni o classi) identificati nel disegno e vengono descritte le loro specifiche funzionalità, l'algoritmo utilizzato, il significato delle variabili passate come parametri, le eventuali variabili globali utilizzate, le variabili locali di cui fa uso e quelle di particolare significatività e i singoli casi d'uso.
Viene quindi allegato il codice sorgente e il modello logico del database.

Piano delle prove

Definiamo dapprima con batteria di prove una sequenza di singoli casi di prova correlati che prevede la verifica di una funzionalità del sistema in corrispondenza di determinati input, che deve essere così documentata:

- classificazione della prova: identificativo, titolo, elenco delle componenti coinvolte nella verifica, descrizione degli obiettivi in termini di funzionalità, livello d'importanza della prova.

- istruzioni per il verificatore: descrizione dei dati in ingresso, singoli o di sequenze particolari;
- risultati attesi: risultati attesi e/o comportamenti in casi estremi;
- istruzioni di ripristino: modalità di recovery in caso di fallimento della prova.

Il piano delle prove è un documento che può essere articolato in due sezioni:

- definizione delle prove di integrazione: generalmente il sistema deve interagire con altri preesistenti, sia interni che esterni, e quindi devono essere pianificate le batterie di prove e descritta la strategia di integrazione; devono essere descritti i driver e gli stub utilizzati per le prove di modulo e per le prove condotte a ogni passo d'integrazione, eventualmente allegando i codici sorgenti;
- definizione delle prove di collaudo (del sistema completo): dopo aver verificato singolarmente le unità funzionali del sistema è necessario pianificare le batterie di prove per la verifica delle funzionalità nel suo complesso e il soddisfacimento dei requisiti software prefissati.

Il collaudo termina con la stesura del documento di accettazione nel quale il cliente sottoscrive il suo soddisfacimento per il prodotto che gli è stato consegnato.

Risultati delle prove

Per ogni caso/batteria di prove viene prodotto un report da parte del verificatore che riporta la data del test, il tipo di prova identificato con il riferimento univoco, i dati di prova utilizzati e l'esito, commentando eventuali risultati inattesi e situazioni indesiderate ed eventualmente fornendo le indicazioni per una eventuale ripetizione della prova.

Manuale d'uso (o manuale utente)

Il manuale utente ha lo scopo di mettere l'utente in grado di utilizzare il programma senza problemi e deve essere redatto in un linguaggio semplice, di facile comprensione per ciascun utente del sistema, tenendo conto che sicuramente ci saranno persone senza conoscenze informatiche approfondite.

> Disegni, schemi e sequenze guidate nella descrizione delle operazioni agevolano la loro comprensione: più la documentazione è chiara ed esaustiva e meno gli utenti telefoneranno per richiedere assistenza operativa!

La documentazione utente comprende:

- documentazione cartacea:
  - manuale d'uso;
  - manuale di installazione, se il prodotto viene pacchettizzato e installato dal cliente;
- documentazione elettronica:
  - help on-line;
  - eventuale sito di riferimento.

Sia il formato cartaceo che quello elettronico devono contenere le sezioni seguenti:

- introduzione: dopo una breve descrizione dell'utilizzo del manuale vengono descritti i servizi forniti dal programma e l'ambiente hardware/software in cui il sistema funziona (tipo di calcolatore, memoria di massa e di lavoro necessarie, tipo e versione del sistema operativo, tipo e versione del linguaggio di programmazione utilizzato) e una descrizione delle modalità di avvio del programma;
- istruzioni d'uso: la descrizione di ogni singola funzione e di ogni singola voce del menu (se esiste), possibilmente integrata con un esempio guidato dove vengono elencate le singole azioni richieste (e permesse) all'utente; per ogni caso devono essere elencate le possibili situazioni di errore con l'elenco delle cause e l'indicazione dei possibili interventi necessari alla loro risoluzione;
- appendice: viene riportato il glossario, l'elenco dei messaggi di errore comuni a tutto il sistema, una sezione di come risolvere le principali situazioni critiche.

Tool di documentazione

Per aiutare gli sviluppatori a produrre la documentazione del codice sono presenti tool di documentazione automatica che analizzano il codice individuando appositi contrassegni (tag) predisposti dal programmatore.
Come vedremo nella prossima lezione, l'aggiunta di commenti all'interno del codice è una metodologia di documentazione interna che viene effettuata per facilitarne la lettura, per spiegarne il funzionamento e per favorire la collaborazione tra diversi sviluppatori: utilizzando una notazione specifica per questi commenti si ottiene il doppio beneficio di commentare il codice predisponendo gli elementi necessari per poi ottenere automaticamente la generazione della documentazione dei programmi sorgenti.

> È però necessario porre molta attenzione per rispettare gli standard previsti dal generatore automatico che si intende utilizzare, sia nella forma che nella completezza necessaria per produrre un documento esaustivo, chiaro e ben dettagliato.

È consigliabile produrre questa documentazione durante lo sviluppo, anche perché la persona più indicata per descrivere un programma è proprio chi lo sta scrivendo: in questo modo il documento viene costantemente aggiornato, in linea con l'evoluzione del software e risulta pronto "contemporaneamente" col codice da collaudare.
Può quindi essere utilizzato dai collaudatori durante le operazioni di test, permettendo di risparmiare notevoli risorse, sia in termini di tempo che naturalmente di denaro.

> Nessun programmatore ama scrivere relazioni e documenti: avere uno strumento di generazione automatica delle documentazione del codice lo libera da un compito "noioso" e poco gratificante e gli permette di utilizzare "meglio" il proprio tempo per migliorare il software che sta scrivendo.

Alcuni tool permettono di personalizzare lo "strumento di generazione automatica" arricchendo l'insieme dei tag in modo da ottenere una documentazione più approfondita, completa e aderente ai fabbisogni del team di sviluppo: è infatti possibile aggiungere elementi specifici per i collaudatori affinché possano effettuare le varie fasi di test con riferimenti a esempi concreti.

Tool esistenti

I principali tool appositamente sviluppati per effettuare la generazione automatica della documentazione sono Javadoc e Doxygen.

- Javadoc è stato sviluppato direttamente dalla Sun Microsystems nel 1990 durante lo sviluppo del linguaggio Java per documentare proprio la sua progettazione e venne distribuito insieme al kit di sviluppo per la documentazione di progetti scritti in questo linguaggio.
  Col passare degli anni il tool è cresciuto assieme al linguaggio, divenendo sempre più ricco e completo, e aggiungendo la possibilità per l'utilizzatore di personalizzare il contenuto e il formato di output della documentazione.
- Doxygen è un sistema multipiattaforma per la generazione di documentazione tecnica di un qualsiasi prodotto software e fu sviluppato come progetto *Open Source* a partire dal 1997 su proposta di Dimitri van Heesch.
  A differenza di Javadoc utilizzabile solo per Java, Doxygen tool può essere utilizzato con codice sorgente scritto in diversi linguaggi: C++, C, Java, Objective C, Python, IDL, PHP e C#.

> Sia Doxygen che Javadoc analizzano il codice sorgente ed estraggono informazioni dalla dichiarazione di strutture dati e da commenti scritti con una particolare sintassi, generando un ipertesto o un formato HTML oppure anche un formato pdf: li descriveremo entrambi.

Verifichiamo le conoscenze

Esercizi a scelta multipla

1 Quale tra i seguenti non è uno standard utilizzato nella produzione della documentazione?

a) standard per la produzione di un documento
b) standard per la manutenzione di un documento
c) standard per la pubblicazione di un documento
d) standard per la distribuzione di un documento

2 Quale tra i seguenti non è uno standard di manutenzione di un documento?

a) l'identificazione di un documento
b) l'editing di un documento
c) l'aggiunta di un documento
d) l'aggiornamento di documenti

3 La documentazione di un sistema riguarda anche (indicare quello inesatto):

a) le specifiche funzionali e non funzionali del sistema
b) le scelte di progetto riguardanti l'architettura generale
c) lo studio di fattibilità
d) le strutture degli archivi e dei dati in memoria centrale e i principali algoritmi utilizzati;
e) le modalità d'uso del programma
f) i collaudi effettuati e i loro risultati.

4 Quale tra i seguenti documenti non appartiene alla documentazione del management del progetto?

a) organigramma
b) diario di progetto
c) verbale
d) analisi del dominio
e) norme di progetto
f) offerta
g) contratto per lo sviluppo, la fornitura del software e l'assistenza all'avviamento
h) piano di gestione della qualità
i) relazione finale

5 Quale tra i seguenti documenti non appartiene alla documentazione del progetto?

a) analisi dei requisiti (documento SRS)
b) specifica architetturale
c) specifica di dettaglio
d) piano di progetto
e) piano delle prove
f) risultati delle prove
g) manuale d'uso

6 Il piano del progetto contiene le attività pianificate, in particolare (indicare quello non presente):

a) la definizione degli obiettivi
b) l'analisi dei rischi
c) la descrizione del modello di processo di sviluppo e delle singole fasi
d) la stima dei costi
e) l'attività di progetto (Work Breakdown Structure, Diagramma di Gantt)
f) il piano dei collaudi
g) il consuntivo attività
h) gli strumenti utilizzati

Test vero/falso

1 La documentazione è un supporto al processo di sviluppo del software. V F
2 La documentazione quindi deve essere redatta durante lo svolgimento del progetto. V F
3 La documentazione di un sistema riguarda soltanto il codice sorgente. V F
4 L'offerta rientra tra la documentazione inerente il management del progetto. V F

5 Il piano di gestione della qualità non è tra i documenti del management del progetto. V F
6 Tra le norme di progetto rientra l'elenco degli strumenti utilizzati. V F
7 Nel contratto per lo sviluppo sono presenti i tempi di consegna e le eventuali penali. V F
8 Col contratto per lo sviluppo viene proposto al cliente un contratto di manutenzione. V F
9 Alla relazione finale viene generalmente allegato il manuale operativo. V F
10 Nel manuale utente sono presenti gli esiti delle prove delle funzionalità del prodotto. V F

Esercizi di completamento

1 La documentazione del progetto si compone di tre parti:
a) .....
b) .....
c) .....

2 Possiamo classificare la documentazione di un progetto secondo quattro modalità:
a) ..... o .....:
- .....: separata dal programma vero e proprio;
- .....: sotto forma di commenti o help contenuti nel file sorgente;

b) ..... o .....:
- .....: richiamabile su richiesta durante l'esecuzione del programma,
- .....: sotto forma di manuale consultabile separatamente;

c) ..... o .....:
- .....: cioè riguardante il programma nel suo complesso;
- .....: sotto forma di commenti inseriti nel codice sorgente in punti specifici del programma;

d) ..... o .....:
- .....: per chi usa il programma
- .....: è specifica per il personale tecnico interessato alla struttura del programma per lo sviluppo o per la sua manutenzione

3 La documentazione rivolta all'utente è generalmente organizzata in:
a) .....
b) .....
c) .....

4 La documentazione utente comprende:
a) documentazione .....:
- manuale .....;
- manuale .....;

b) documentazione .....:
- .....;
- eventuale .....

LEZIONE 2

LA DOCUMENTAZIONE DEL CODICE

IN QUESTA LEZIONE IMPAREREMO...

- il concetto di documentazione interna
- naming, indentazione e commenti

Generalità

Documentare il codice è una attività fondamentale che rientra nella documentazione di un progetto software e quindi, oltre alla documentazione a corredo di tutte le fasi del ciclo di vita del software, è fondamentale anche la documentazione interna del codice.

Un programmatore produce centinaia di programmi, solitamente lavorando in un gruppo di sviluppo in collaborazione con altri professionisti, e nello sviluppo di un sistema lo stesso segmento di codice deve essere "ripreso in mano" più volte, sia dall'autore che da altri componenti del gruppo di lavoro.

È necessario che tutto il gruppo di lavoro adotti un insieme di tecniche e di regole di scrittura del codice per stilare una corretta e completa documentazione del software per facilitare e supportare molte fasi del ciclo di vita:

- testing: la leggibilità del codice sorgente agevola la sua verifica e aiuta nella ricerca degli errori;
- reverse engineering e reengineering: permette il riutilizzo del codice in problemi differenti e la cooperazione e scambio di routine tra programmatori;
- integrazione: favorisce la definizione delle interfacce per interagire con i sistemi preesistenti;
- manutenzione: consente la manutenzione rapida ed efficiente del codice anche a distanza di tempo, per aggiungere nuove funzionalità, modificare funzionalità esistenti, correggere gli errori o migliorare le prestazioni.

Occorre perciò stabilire alcune norme di leggibilità nella stesura dei programmi: alcune di queste regole sono frutto dell'esperienza e del buon senso, altre sono imposte dal datore di lavoro e dagli standard di settore, e insieme concorrono a migliorare la qualità del lavoro del team di sviluppo.

La documentazione interna al codice dovrebbe essere:

- coerente: non devono esserci ambiguità o contraddizioni;
- consistente: conforme a uno standard;

- tracciabile: deve essere possibile poter collegare, nella maniera più rapida possibile, i concetti presenti nel codice con la loro documentazione e con i concetti a essa associati.

La definizione delle regole di scrittura del codice deve essere effettuata prima dell'inizio dello sviluppo; nel caso di progetti greenfield engineering si è liberi di usare lo standard usuale utilizzato dalla software house incaricata della realizzazione del prodotto, mentre nel caso di integrazione o manutenzione di software esistente è necessario adeguarsi ai sorgenti che devono essere aggiornati.

> Lo standard adottato deve essere rispettato sempre e da tutti omogeneamente nel corso di tutto il ciclo di sviluppo: il prodotto finale deve "sembrare" tutto sviluppato da un unico programmatore.

Naturalmente lo stesso standard deve essere applicato anche successivamente allo sviluppo, nel corso di revisioni periodiche del codice.

Se un codice è "scritto bene" e di facile comprensione sicuramente è meno soggetto a errori sia in fase di sviluppo che successivamente, quando vi "si deve mettere mano" per le revisioni periodiche.

> L'utilizzo di tecniche di scrittura del codice affidabili e di regole di programmazione valide per la creazione di un codice di alta qualità è di fondamentale importanza per le prestazioni e la qualità del software.

Il codice sorgente

Il codice sorgente deve essere riportato completamente nella documentazione e deve contenere:

- una intestazione in cui deve essere scritto il nome e il cognome dell'autore, la versione del programma e il nome del programma, la data dell'ultima modifica apportata e una breve descrizione delle funzionalità;
- il significato delle strutture dati e delle eventuali variabili globali;
- la separazione tra ogni sottoprogramma/metodo: l'inizio di ogni funzione/metodo deve essere chiaramente individuabile e per ciascuna di esse deve essere indicata sinteticamente la funzionalità, il significato dei parametri, le variabili globali e le variabili di particolare significatività; inoltre, se il codice è particolarmente complesso, deve essere presente la descrizione dell'algoritmo utilizzato in pseudocodifica;
- molti "buoni" commenti al codice, soprattutto in corrispondenza di punti critici o di linee di codice che potrebbero risultare poco chiare.

Le tecniche di scrittura del codice comprendono diversi aspetti dello sviluppo del software, e nella nostra trattazione ne analizziamo i principali che competono alla realizzazione della documentazione interna:

- il naming delle variabili (naming guidelines);
- le regole di scrittura del codice;
- i commenti.

Naming Guidelines

Per Naming Guidelines si intende un insieme di regole da seguire nella scelta dei nomi di variabili, delle funzioni/metodi e di ogni elemento che deve definire il programmatore.

La prima regola è quella di dare a ogni elemento un nome significativo, in modo che "da solo" renda sufficientemente comprensibile il suo utilizzo.
Per esempio, alla variabile che contiene un progressivo è meglio dare il seguente identificatore:

```
int sommaMensileIvaDare
```

piuttosto che

```
int valoreIva
```

Anche la forma di scrittura degli indentificatori è importante, e i due casi sono definiti come

PASCAL E CAMEL CASED

Si definiscono Pascal Cased i nomi che hanno le iniziali maiuscole, per esempio CalcoloFattura; sono Camel Cased i nomi che hanno l'iniziale della prima parola minuscola e le iniziali delle rimanenti parole maiuscole, per esempio totaleFattura.

ESEMPIO 3

Nella programmazione a oggetti è consuetudine dare alle *classi* nomi Pascal Cased mentre agli *attributi* nomi Camel Case, in modo da individuare direttamente "cosa sono".

Notazione ungherese

Con "notazione ungherese" si indica la proposta naming convention fatta da Charles Simonyi per i nomi delle variabili nella quale ogni nome di variabile dipende dal suo tipo e dal suo scopo.

Ogni nome di variabile inizia con un prefisso costituito da una o più lettere minuscole in modo da formare un codice mnemonico per il tipo o lo scopo di questa variabile: il primo carattere del nome assegnato è in maiuscolo per separarlo visivamente dal prefisso, coerentemente con lo stile CamelCase.

ESEMPIO 4

Tipo variabile	Prefisso	Esempio
flag booleano	f	fErrore
char	ch	chScelta
stringa	str	strNome
puntatore	ptr	ptrTesta
integer	int	intSomma

Questa notazione è utile al programmatore come elemento di documentazione della tracciabilità in quanto, ogni volta che viene utilizzata una variabile, il programmatore è consapevole del suo tipo e nei linguaggi a oggetti si evitano errori semantici dovuti a errata interpretazione del tipo, in caso di polimorfismo.

La notazione ungherese, accolta con entusiasmo nel momento in cui fu presentata al "mondo informatico", oggi viene poco utilizzata in quanto aumenta la dimensione del codice e il tempo necessario a scriverlo e può portare a problemi di errata interpretazione degli acronimi dei tipi per le classi e per i tipi definiti dall'utente.
Microsoft, nelle MSDN, per esempio, inizia a sconsigliarne l'utilizzo, precedentemente consigliato.

Rimane comunque un valido esempio che può essere personalizzato dal team di sviluppo, eventualmente usato in modo parziale e solo per i linguaggi di scripting lato client e nei moderni linguaggi object oriented.

Naming Convention Microsoft

Microsoft, uno tra i maggiori produttori di software al mondo, ha indicato un insieme di regole e di consigli per favorire nel migliore dei modi la comprensione del flusso logico di un'applicazione. Tra tutti ricordiamo i tre riportati di seguito:

A il nome che viene assegnato a un elemento deve indicarne la sua funzione e non le modalità con cui questa viene eseguita: infatti si potrebbero effettuare in seguito rettifiche nel codice che potrebbero portare a situazioni di incoerenza.

ESEMPIO 5

Alla funzione/metodo che restituisce il prossimo cliente è preferibile dare il nome

```
GetNextCliente()
```

piuttosto che

```
GetNextArrayElement()
```

In questo modo si mantiene un alto livello di astrazione, anche perché successivamente, al posto dell'array, si potrebbe utilizzare un file o una tabella di un database e il nome della funzione trarrebbe in inganno.

B Anche la lunghezza del nome deve essere valutata, utilizzando nomi sufficientemente lunghi in modo che siano comprensibili, ma allo stesso non troppo dettagliati: spesso è difficile dare il nome adatto che sia un giusto compromesso che soddisfa ogni esigenza.

C Bisogna sempre tener presente che i nomi devono essere significativi per tutti coloro che dovranno leggere il codice, e non solo per chi lo ha scritto: è necessario mantenere sempre la conformità dei nomi scelti agli standard e accertare le regole del linguaggio in uso.

Descriviamo singolarmente le tipologie più importanti, cioè variabili e funzioni.

Variabili e costanti

Le variabili vanno scritte in minuscolo mentre le costanti vanno scritte interamente in MAIUSCOLO.

È opportuno utilizzare una notazione simile a quella ungherese per identificare l'utilizzo della variabile, cioè con l'aggiunta di un suffisso:

Tipo variabile	Suffisso	Esempio
contatore	Cnt	numeriCnt
media	Avg	speseAvg
somma	Sum	fattureSum
minimo	Min	altezzaMin
massimo	Max	pesoMax
indice	Index	tabellaIndex

I nomi sono sempre composti e quindi si utilizza la combinazione di caratteri maiuscoli e minuscoli per semplificarne la lettura, adottando la convenzione notazione dell'OOP, cioè il sistema Camel per i nomi di variabile (per esempio ivaMensileClienti) con la prima iniziale minuscola.

Gli **acronimi** costituiscono una eccezione, poiché vengono scritti in maiuscolo se formati da 2 caratteri, altrimenti in Pascal Cased:

```
public int ID
public string Html
public float Iva
public string Cap
```

È vivamente consigliato non definire più variabili sulla stessa riga al fine di lasciare spazio per aggiungere un breve commento per ciascuna di esse.

Nei nomi delle variabili booleane si aggiunge il prefisso "Is", "Has" o "Can" in modo che implichino automaticamente la domanda e quindi il contenuto Vero/Falso:

```
fileIsOpen
recordIsFound
```

Evitare di usare variabili composte da una singola lettera: solo nei contatori dei cicli è ammesso utilizzare caratteri singoli, ma evitando i o j facilmente confondili con 1 (uno) ed l (elle).

Non utilizzare *"numeri magici"* ma definire sempre le costanti che contengono tali valori per favorire sia la loro comprensione che i possibili utilizzi futuri con dimensioni diverse:

```
int x = 0; x < 10; x++)                  // codifica non ottimale
int x = 0; x < dimVettore; x++)          // codifica corretto

int k = 0; k < 7; k++)                   // codifica non ottimale
int k = 0; k < giorniSettimana; k++)     // codifica corretto
```

Persino la più ragionevole costante cosmica cambierà un giorno o l'altro: pensate che vi siano 365 giorni in un anno? I vostri clienti su Marte saranno molto infastiditi dal vostro sciocco pregiudizio!

ESEMPIO 6

Create una costante

```
… int DAYS_PER_YEAR = 365;
```

così potrete produrre agevolmente una "versione marziana" del vostro prodotto senza dover sostituire per tutto il vostro codice i ◀ numeri magici ▶ inseriti.

◀ Numero magico Un *numero magico* è una costante numerica integrata nel codice, senza usare una definizione di costante. Qualsiasi numero, a eccezione di -1, 0, 1 e 2, è considerato *magico*. ▶

Routine\funzioni\metodi

La prima regola valida anche per i nomi delle funzioni è quella di evitare nomi poco chiari e suscettibili di interpretazioni soggettive, che possono solamente creare ambiguità.
Anche i nomi delle funzioni sono sempre composti e quindi si utilizza il sistema Pascal (per esempio `CalcolaIvaTotale`) con tutte le iniziali in maiuscolo.

Nel caso di classi è inutile dare ai metodi nomi già impliciti in quello delle classe:

```
Libro.PagineLibro            // naming non ottimale
Libro.Pagine                 // naming corretto
```

Nel caso di funzioni/metodi che eseguono specifiche operazioni sugli oggetti è opportuno utilizzare la struttura "verbo – nome", come negli esempi riportati in seguito:

```
… SaldoIva()                 // naming non ottimale
… CalcoloSaldoIva()          // naming corretto

MinimoElemento()             // naming non ottimale
RicercaElementoMin()         // naming corretto
```

Utilizzare il prefisso "Is", "Has" o "Can" nel caso di metodi che restituiscono un boolean

```
boolean IsPresent()
boolean HasRecord()
boolean CanReadFile()
```

Definire le variabili di controllo per i cicli non tutte assieme all'inizio del codice, ma appena prima di utilizzarle, per non correre il rischio di "trovarle" con valori indesiderati:

```
boolean avanti = true;
while ( avanti )
{
 . . .
}
```

Nei linguaggi OOP o comunque in tutti quelli che ammettono l'overload delle funzioni occorre stabilire uno standard di denominazione che consenta di associare le funzioni simili; è inoltre necessario che tutti gli overload eseguano una funzione simile.

È meglio non scrivere funzioni/metodi più lunghi di 20 linee: in tali casi è sicuramente scomponibile la funzione in due o più parti richiamabili in sequenza.

File

Nella suddivisione in file del proprio lavoro, è bene evitare nomi come:

```
Media ponderata con il secondo metodo matematico di Eulero.c
ciro&pippo.c
prova I/O.c
funzione23.c
```

Usare invece nomi che ricordino il lavoro svolto in modo sintetico:

```
Media_Aritmetica.c
Primitive_Albero.c
Gestione_USB.c
Ordinamenti.c
```

L'uso di nomi corti, senza spazi e senza caratteri speciali o accentati, oltre a facilitare la manipolazione dei file, garantisce la compatibilità con altri sistemi operativi e protocolli di trasferimento: ricordiamo che per esempio il protocollo Joliet utilizzato dai CD-ROM consente al massimo 31 caratteri per i nomi dei file.
È sempre necessario tener traccia delle successive revisioni del software, come vedremo in seguito: un semplice metodo per poter risalire alla precedente versione è quello di numerare progressivamente i file con due cifre che indicano il numero di versione:

```
Media_Aritmetica_04.c
Primitive_Albero_33.c
Gestione_USB_22.c
```

Commenti

Il software viene descritto nella documentazione esterna ma è necessario che i listati di codice sorgente siano ben commentati e siano facilmente comprensibili anche da soli, senza fare rifermento a manuali cartacei.
Spesso la scrittura dei commenti non avviene contemporaneamente a quella delle istruzioni: si preferisce "prima terminare" un segmento", testarlo e collaudarlo, e solo al termine commentarlo; è invece importante iniziare proprio con la scrittura dei commenti, indicando a parole l'algoritmo in "pseudocodifica" e successivamente trasformare proprio queste righe in commento. Questo metodo di procedere è il più indicato soprattutto per moduli particolarmente complessi.
La quantità di commenti deve superare abbondantemente il codice: un altro programmatore deve capire come funziona una routine senza chiedere nulla a chi l'ha scritta.

Il valore del riferimento a un oggetto non può essere modificato esplicitamente all'interno del codice di un programma in quanto Java ne maschera il contenuto.

Vediamo un elenco di semplici consigli:

A ogni programma/classe deve essere preceduto da un commento predefinito, uguale per tutto il codice, che indichi brevemente le note di copyright, l'autore, la data e la versione, la funzionalità svolta;

```
/**
COPYRIGHT (c) 2013 MySoft snc. All Rights Res.
Classe per l'interfacciamento con gli Smartphone
@author Paolo Rossi
@version 1.01 2013-12-15
*/
```

B ogni funzione deve essere preceduta da un commento predefinito, uguale per tutto il codice, che indichi brevemente la funzionalità svolta, il valore di ritorno, i parametri in ingresso, i limiti di funzionamento e i casi particolari:

```
/**
Converte una data del calendario attuale in quello giuliano
Nota: algoritmo tratto da:        Numerical Recipes in C,2nd ed.
Cambridge University Press, 1992
Limiti di funzionamento: calcola solo date dopo l'anno 1000
@param day      giorno della data da convertire
@param month    mese della data da convertire
@param year     anno della data da convertire
@return         il numero del giorno del calendario giuliano
*/
```

C preferire un commento su linea singola piuttosto che di seguito alle istruzioni; solo per descrivere il contenuto delle variabili è opportuno affiancare il commento, che deve essere incolonnato agli altri:

```
double xold;            // vecchia ascissa
double xnew;            // nuova ascissa interpolata
```

D non inserire caratteri grafici e righe inutili di simboli (tipo righe di asterischi) solo per abbellire il commento: è meglio spaziare bene i commenti eventualmente lasciando righe bianche, che meglio si prestano a essere utilizzate come vedremo dai generatori automatici di documentazione;

E è inutile "*commentare bene un codice scritto male*": se si fa fatica a descrive un segmento di codice perché è troppo lungo o complesso, probabilmente merita di essere riscritto!

F i commenti devono essere chiari e precisi, senza lasciare adito ad ambiguità e non devono essere prolissi; non si devono inserire commenti inutili o ovvi, che recano solo disturbo alla lettura del codice senza portare informazione: se un commento non serve è "inutile metterlo"!

```
giorni = giorni + 1;    // aggiungo 1 alla variabile giorni
giorni = giorni + 1;    // se l'anno è bisestile si aggiunge un giorno
```

G è opportuno inserire un commento in prossimità delle condizioni logiche che sono di fondamentale importanza per la comprensione del flusso di esecuzione dell'algoritmo: descrivere i casi di ingresso/uscita e le eventuali situazioni particolari;

- quando un commento va su due righe è meglio non utilizzare due righe con commento //: risulta scomodo in seguito dovere eventualmente spostare entrambe le righe per mantenere l'indentamento del codice:

```
// commento - non fate in questo modo
// … altra riga commento precedente
// … segue ancora commento
```

- quando si utilizza un commento che va su più righe non si devono aggiungere caratteri grafici inutili:

```
/* commento - non fate in questo modo
* altro commento
* ancora commenti
*/
```

l'aspetto sarà anche gradevole, ma costituisce un forte disincentivo ad aggiornare il commento.

> Qualcuno utilizza editor di testi che dispongono automaticamente i commenti con dei formati grafici predefiniti: se si opta per questa possibilità è bene assicurarsi che la prossima persona che farà interventi di manutenzione sul vostro codice disponga di un editor di quel genere.

Disporre i commenti lunghi in questo modo:

```
/*
commento
altro commento
ancora commenti
*/
// oppure nel seguente modo
/*
commento
altro commento
ancora commenti
*/
```

Formattazione

Anche la formattazione del codice è fondamentale per favorire la comprensione dei sorgenti e rientra nella documentazione interna; di seguito riportiamo alcuni suggerimenti per effettuare una "buona codifica", facilmente adattabili a ogni linguaggio di programmazione.

> Per prima cosa è necessario stabilire una lunghezza massima di riga per i commenti e per il codice in modo da poter visualizzare interamente la riga senza dover scorrere a destra e a sinistra la finestra di editor.

Spaziature

È bene definire il numero di caratteri utilizzati per l'indentazione (due o tre, non di più altrimenti non si lascia spazio per i commenti in linea!) e mantenerli costanti per tutta le stesura del codice: è

sconsigliato l'uso dei tabulatori per inserire spaziature in quanto potrebbero essere diversi a seconda dell'editor utilizzato e quindi portare a disallineamenti nel corso della manutenzione.
Lasciare sempre uno spazio vuoto prima e dopo gli operatori, tranne i casi in cui questa notazione alteri la funzione scelta (come per esempio per i puntatori in C++).

```
x1=(-b-Math.sqrt(b*b-4*a*c ))/( 2*a);                  // pessimo
x1 = (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);    // ottimo
```

Lasciare uno spazio vuoto dopo (e non prima) ogni virgola e punto e virgola: non lasciare invece uno spazio prima o dopo una parentesi tonda o quadra.

```
for (i = 0, j = N; i < j && j > 10; i ++, j --)
   if (A[i][j])
      A[i][j] *= i;
//è sicuramente più chiaro di
for(i=0,j=N;i<j&&j>10;i++,j--)if(A[i][j])A[i][j]*=i;
```

Istruzione if

L'istruzione if annidata può portare a errate interpretazioni, quindi è opportuno indentarla con cura:

```
// codifica if annidati incomprensibile
if ( ... )
...
if ( ... )
...
else
...
else
...

// codifica if annidati comprensibile
if ( ... )
   if ( ... )
   ...
   else
   ...
else
...
```

Evitare la trappola "`if ... if ... else`" aggiungendo le parentesi anche se non necessarie:

```
// codifica con dandling-else

if ( ... )
   if ( ... )
   ...;
else
...;
```

```
// caso a: else riferito al primo if
if ( ... )
{
   if ( ... )
   ...;
}                          // le graffe {...} sono necessarie
else
...;

// caso b: else riferito al secondo if
if ( ... )
{
   if ( ... )
   ...;
   else
   ...;
}                          // le graffe non sono necessarie, ma evitano guai
```

Parentesi graffe

Le parentesi graffe di apertura e di chiusura si devono allineare **orizzontalmente** solo se è presente una singola istruzione:

```
while (x < tanti){ ... }
for (x = 0; x < tanti; x++){ ... }
```

oppure **verticalmente**:

```
while (x < tanti)
{
   ...
}
for (x = 0; x < tanti; x++)
{
   ...
}
```

Evitare di mettere la parentesi graffa aperta dopo la parola chiave:

```
while (x < tanti){         // non fatelo
   ...
}
```

perché in questo modo si rende più difficile la verifica della corrispondenza delle parentesi graffe.

Tool di indentazione automatica: JIndent

Esistono prodotti che effettuano l'indentazione automatica del codice scritto nei più comuni linguaggi di programmazione: uno di questi è Jindent che esegue l'elaborazione del codice sorgente sia Java che C++ offrendo al programmatore un insieme di possibilità di settaggi (oltre 300 opzioni) che possono essere configurate per meglio realizzare un codice nello stile desiderato secondo le convenzioni adottate dal team di sviluppo.

Jindent è un programma gratuito per uso non commerciale, scaricabile dal sito http://www.jindent.com.
Tra le operazioni svolte automaticamente da Jindent, ricordiamo l'inserimento e la sostituzione dell'header e del footer, la formattazione automatica degli spazi e delle righe bianche, la conversione dei caratteri minuscolo/maiuscolo, l'inserimento delle parentesi e il loro incolonnamento ecc.

◀ **Non-commercial/Academic License** This license type may be used for non-commercial purposes only. We define non-commercial purposes as non-profit educational purposes, free software development and purposes that are not intended to be commercially exploited. ▶

Sono anche disponibili i plugins di Jindent per i più diffusi ambienti di sviluppo IDE, come Eclipse, Netbeans, JDeveloper, IntelliJ IDEA, JBuilder.

Riportiamo di seguito la descrizione di alcune delle funzionalità offerte da Jindent, così come vengono descritte sul sito http://www.jindent.com.

◀ **Jindent** Jindent is the leading commercial solution among all source code formatter tools and has proven its functionality and stability worldwide.
Jindent automatically formats all source code files for you according your corporate code convention: ***don't waste time by formatting code by hand!***
Jindent supports more than 300 formatting settings:

- **Intelligent line wrapping**: Jindent knows several algorithms to perform line wrapping and tries always to find the best fitting solution. The line wrapping algorithms keep attention of operator precedence, prefer low level wrapping, and much more.
- **Scope related indentation**: several options to control indentation of blocks, tokens and scopes by white spaces and tabulators.
- **Brace style transformation**: separated and fine grained control of brace style settings for classes, interfaces, annotation types, methods, constructors, statements, enums and try-catch-blocks. Additionally the most common braces styles are already available as presets: K&R/Kernal, BSD/Allman, Whitesmiths and Gnu.
- **Insertion of parentheses and braces**: to improve readability of source code Jindent is able to insert parentheses and braces into certain statements and conditions without changing its functionality.
- **Blank line and white space formatting**: support for very fine grained white space and blank line control before and after tokens.
- **Semantic source code separation**: automatical insertion of blank lines and special comments to separate source code elements by its semantic.
- **Sorting of source code elements**: very fine graind sorting options for: fields, initializers, annotation types, constructors, methods and import declarations. Sorting keys can be modifier, types, names, parameters count, assignments and many more.
- **Javadoc validation, formatting and template-driven generation**: JavaDocs can be automatically validated for parameters, exceptions, return types and generics. Missing JavaDoc tags or completely missing JavaDocs can be inserted by customizable templates. Obsolete JavaDoc tags can be removed.
- **Insertion and substitution of header and footer**: Jindent is able to recognize header and footer comments. Missing headers/footers will be inserted or obsolete headers/footers will be recognized and substituted by new ones.
- **Conversion between character and end-of-line encodings**: support of a lot of different encodings and text styles to read and write source codes. Therefore it is possible to convert Java files from one encoding standard to another.
- **Understands Java 7.0 source code including SQLJ-Statements**: formatting support is enabled for all kind of Java files (Java 7.0 syntax and below) including SQLJ-statements.
- . . .

<estratto della documentazione presente su http://www.jindent.com/> ▶

Verifichiamo le conoscenze

Esercizi a scelta multipla

1 La documentazione del software facilita e supporta le fasi (indicare quella non compresa):

a) testing
b) reverse engineering
c) reengineering
d) analisi dei requisiti
e) integrazione
f) manutenzione

2 La documentazione interna al codice deve essere (indicare quella inesatta):

a) coerente
b) aderente
c) consistente
e) conforme a uno standard
e) tracciabile

3 I principali aspetti dello sviluppo del software che competono alla realizzazione della documentazione interna sono

a) il naming delle variabili
b) le regole di scrittura del codice
c) la scomposizione in moduli
d) i commenti

4 Quali tra i seguenti metodi sono correttamente scritti secondo la convenzione Camel Cased?

a) CalcolaTotale
b) TOTFattura
c) Somma_età
d) getValore
e) media_voti
f) Media_Voti

5 Quali tra le seguenti variabili sono correttamente scritte secondo la convenzione Pascal Cased?

a) totaleIva
b) TOTFattura
c) Somma_età
d) valoreMax
e) media_voti
f) meseIVA
g) Media_Voti

6 In base agli standard Microsoft, interpreta i seguenti Naming accoppiando il significato

1 ..... metodo
2 ..... classe
3 ..... variabile
4 ..... costante
5 ..... variabile booleana

a) IVA
b) CC
c) iva
d) isMaggiore
e) IsMaggiore

Test vero/falso

1 La documentazione del software è importante solo nel caso di progetti greenfield engineering. V F
2 Il codice sorgente deve essere riportato completamente nella documentazione. V F
3 Per Naming Guidelines si intende un insieme di regole per la scelta dei nomi di variabili. V F
4 Si definiscono Pascal Cased i nomi che hanno le iniziali maiuscole. V F
5 Si definiscono Camel Cased i nomi che hanno le iniziali maiuscole. V F
6 Con "notazione ungherese" si indica la proposta fatta da Simonyi per i nomi delle funzioni. V F
7 La "notazione ungherese" utilizza lo stile Camel Cased. V F
8 Gli acronimi vengono scritti in maiuscolo se formati da almeno due caratteri. V F
9 È consigliabile definire più variabili sulla stessa riga se sono dello stesso tipo. V F
10 I nomi dei file devono essere corti, spaziati e senza caratteri speciali o accentati. V F

Esercitazioni di Laboratorio 1

LA DOCUMENTAZIONE AUTOMATICA CON JAVADOC

Javadoc è stato realizzato direttamente dalla Sun Microsystems nel 1990, durante lo sviluppo del linguaggio Java, per documentare proprio la sua progettazione e, alla fine di essa, venne distribuito insieme al kit di sviluppo per la documentazione di progetti scritti in questo linguaggio.
Nel passare degli anni il tool è cresciuto assieme al linguaggio, divenendo sempre più ricco e completo, e aggiungendo la possibilità all'utilizzatore di personalizzare il contenuto e il formato di output della documentazione.

Generazione Automatica di Documentazione Tecnica

Javadoc è uno strumento che permette di documentare i sorgenti di un programma all'interno dei sorgenti stessi: anziché scrivere la documentazione di un programma in un file separato, il programmatore inserisce nel codice sorgente dei commenti con una particolare sintassi, che verrà mostrata in seguito.
Tali commenti vengono estratti dal programma Javadoc che li converte in un formato più semplice per la consultazione (generalmente in formato HTML); viene effettuato un parsing dei file sorgenti, ovvero un'analisi di tale codice, tramite il compilatore Java.

Il Javadoc può essere eseguito su interi package, su singoli file o su entrambi.

Nel caso di documentazione di un package è possibile memorizzare in un'apposita cartella (una sottocartella che deve avere il nome "doc-files") altri file che possono essere importati nella documentazione, come immagini, esempi di codice o altri file HTML.

Avviando l'esecuzione del ◀ **parser** ▶ del codice di Javadoc, viene analizzato il file sorgente, individuati i tag specifici che identificano i commenti e generato un ipertesto HTML di una o più pagine che descrivono package, classi, interfacce, costruttori, metodi e attributi.

◀ **Parser** A natural language parser is a program that works out the grammatical structure of sentences, for instance, which groups of words go together (as "phrases") and which words are the subject or object of a verb. ▶

Nella documentazione sono presenti sezioni riepilogative, pagine di indice, l'elenco dei metodi e altri strumenti per facilitarne la consultazione.

Il processo di generazione di Javadoc consiste in tre semplici fasi:

- l'inserimento dei commenti di spiegazione del codice sorgente che il programmatore esegue manualmente contemporaneamente alla scrittura dello stesso;
- la predisposizione di eventuali file da associare al documento, inserendo gli opportuni comandi nelle sezioni dove vuole che questi vengano inseriti;
- la generazione vera e propria dell'ipertesto, tramite il comando Javadoc.

ESEMPIO 7

Per generare la documentazione della classe Prova,java all'interno della cartella è sufficiente digitare al prompt dei comandi la seguente istruzione:

```
javadoc Prova.java –d doc
```

Sono molteplici le possibilità offerte come opzione per la compilazione della documentazione che, oltre che sul manuale di riferimento presente sul sito della SUN, sono ottenibili direttamente dalla linea di comando digitando semplicemente Javadoc?.

Il generatore di documentazione è stato sviluppato specificamente per Java, ma con alcuni semplici accorgimenti e adattamenti è possibile utilizzarlo anche per la documentazione di altri linguaggi di programmazione, come per esempio dei sorgenti C++.

Realizzazione di un Javadoc

Vediamo la sintassi dei più comuni comandi: in generale un commento Javadoc è un testo HTML racchiuso tra i tag /** e */.
Scriviamo un esempio di classe Java utilizzando come editor Bluej che è particolarmente comodo per apprendere Javadoc in quanto permette di avere una anteprima della documentazione direttamente dalla finestra di editor del codice sorgente.

ESEMPIO 8

Selezionando Documentation viene mostrata una anteprima della documentazione, dove ritroviamo la nostra frase: ▶

La documentazione minima dovrebbe comprendere la descrizione di ciascun package, classe, interfaccia, metodo pubblico, attributo pubblico e quindi va inserito un commento prima di ciascuno di essi.

All'interno del commento Javadoc abbiamo quindi tre possibili situazioni di commento Javadoc: ▶

```
Monosezione - monolinea :
/** descrizione */

Monosezione - multilinea :
/**
* descrizione
*/

Multisezione - multilinea :
/**
* descrizione
*@tag1 commento per il tag1
*@tag2 commento per il tag2
*/
```

e nell'ultimo è possibile inserire uno o più tag, che iniziano con il simbolo @; riportiamo di seguito i tag più utilizzati.

Tags per documentazione di classi

@author [nome]
Aggiunge "Author:" seguito dal nome specificato: è possibile inserire più autori, presentati in ordine cronologico.

@version [versione]
Aggiunge "Version:" seguita dalla versione specificata.

@see [riferimento]
Aggiunge "See Also" seguito dal riferimento indicato.

Tags per documentazione di metodi

@param [nome del parametro] [descrizione]
Aggiunge il parametro specificato e la sua descrizione alla sezione "Parameters:" del metodo corrente: per ogni parametro va inserito un tag rispettando lo stesso ordine del metodo.

@returns [descrizione]
Aggiunge "Returns:" seguito dalla descrizione specificata indica il tipo restituito e l'insieme dei valori possibili. Non previsto per costruttori e per metodi che non restituiscono alcun valore (void).

@throws [nome completo della classe][descrizione]
Aggiunge "Throws:" seguito dal nome della classe specificata (che costituisce l'eccezione) e dalla sua descrizione. Sono ammessi più tag @exception per ognuna delle eccezioni che compaiono nella sua clausola throws, presentate in ordine alfabetico.

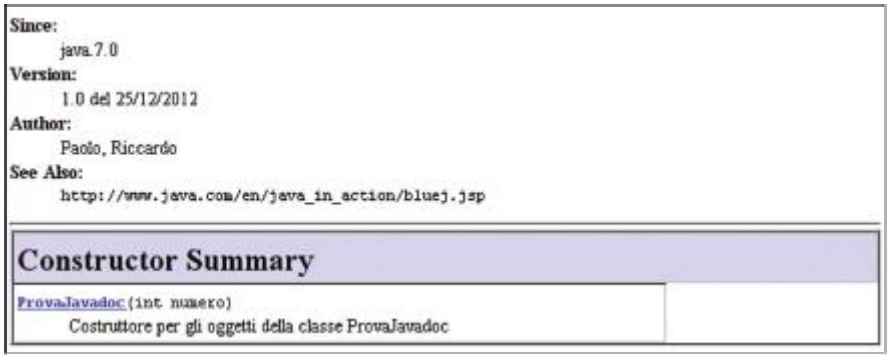
ESEMPIO 9

Inseriamo nel nostro esempio questi tag per commentare un metodo costruttore:

```
/**
 * Questo &egrave; un commento <em>Javadoc</em>.
 * Gli spazi e gli asterischi a inizio riga
 * <strong>sono </strong> sempre ignorati.
 * @author Paolo, Riccardo
 * @version 1.0 del 25/12/2012
 * @see http://www.java.com/en/java_in_action/bluej.jsp
 * @since java.7.0
 */
```

```
public class ProvaJavadoc
{
    /** unico attributo: anni per imparare il linguaggio Java */
     private int anni;

    /**
     * Costruttore per gli oggetti della classe ProvaJavadoc
     * @param numero numero di anni di studio
     */
    public ProvaJavadoc(int numero)
    {
        // inizializza l'attributo della classe
        anni = numero;
    }
```

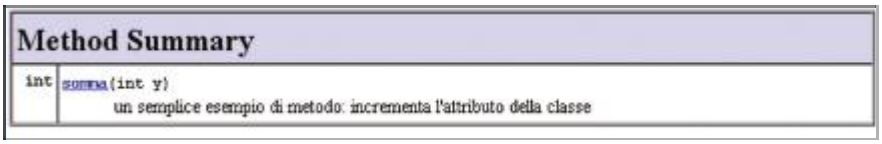
e visualizziamo ora il documento generato:



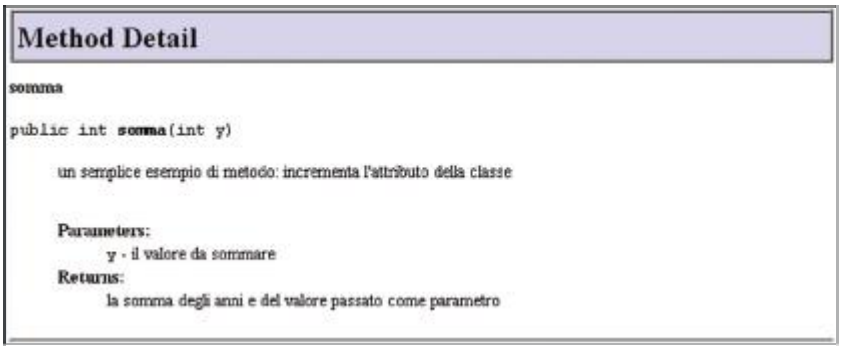
Oltre ai dati dell'intestazione abbiamo l'iperlink al costruttore, che viene riportato in una apposita sezione con il commento da noi inserito: cliccando su di esso veniamo "portati" a una sezione di dettagli del metodo costruttore:

ESEMPIO 10

Aggiungiamo un semplice metodo che somma al nostro attributo un valore numerico e otteniamo un reference nella sezione dedicata ai metodi:



e il dettaglio cliccando sul link.



Nella tabella seguente sono riportati tutti i tag disponibili con una breve descrizione:

Tag	Descrizione
@author	viene seguito da una stringa che identifica l'autore dell'entità documentata
@version	specifica la versione dell'entità documentata, spesso indicata con la versione del software nella quale è stata introdotta; è seguita da un testo descrittivo
@deprecated	identifica un'entità destinata a scomparire nelle future versioni del software ma ancora in uso; viene seguito da una descrizione testuale che solitamente indica l'entità con cui verrà rimpiazzata
@exception @throws	indicano un'entità che può sollevare eccezioni, sono sinonimi e sono seguiti dal nome della classe, ovvero il nome dell'eccezione che può essere generata, e da una descrizione testuale
@param	indica un parametro passato nel costruttore dell'entità documentata; è seguito dal nome del parametro e da una descrizione testuale
@return	specifica cosa può essere restituito come risultato dopo l'esecuzione dell'entità associata; viene seguito da una descrizione testuale
@see	indica un link o un riferimento in relazione con l'entità documentata, può essere espresso in diversi modi: seguito da semplice testo, seguito da link html, oppure, nel caso di riferimenti ad altre entità della documentazione, nella forma "package.classe#metodo testovisualizzato"

Una volta commentati in modo adeguato tutti i sorgenti è possibile specificare le impostazioni per la generazione della Javadoc: prendono il nome di "option" e possono specificare per esempio la presenza della pagina di overview, un CSS diverso da quello standard, frammenti di codice html da porre in cima o alla fine di ogni pagina creata, il titolo del documento, le cartelle o i file sorgenti, la cartella in cui la Javadoc verrà creata, la visibilità o meno di elementi pubblici, protetti o privati.

L'opzione sicuramente più interessante è quella che permette la specifica di una "Doclet" diversa da quella standard: le Doclet sono estensioni di Javadoc che permettono di gestire a piacimento le varie fasi di generazione della documentazione.

ESEMPIO 11

Si supponga di volere la documentazione nel formato PDF: è necessario specificare una doclet in grado di istruire Javadoc come produrre un PDF: la doclet PdfDoclet è una delle più semplici e complete tra le molteplici disponibili (http://pdfdoclet.sourceforge.net/).

Scaricata la libreria (la pdfdoclet-1.0.2-all.jar è disponibile anche tra i materiali della sezione del sito www.hoepliscuola.it riservato a questo volume), è sufficiente specificarne il nome tra le opzioni del comando di generazione:

```
javadoc -doclet com.tarsec.javadoc.pdfdoclet.PDFDoclet
-docletpath pdfdoclet-1.0.2-all.jar -pdf <nome file pdf> prova.java
```

per ottenere un PDF simile a quello di figura:

Per ulteriori opzioni si rimanda alla documentazione specifica di PdfDoclet.

Per queste possibilità e per tutte le modalità di utilizzo del tool Javadoc rimanda alla documentazione di riferimento del "linguaggio" all'indirizzo: http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/index.html

Prova adesso!

Riprendi gli esempi e gli esercizi scritti in Java nelle unità didattiche precedenti e completali con i tag Javadoc creando l'ipertesto completo:

1. Unità didattica 1: lezione 5
2. Unità didattica 1: lezione 6
3. Unità didattica 2: lezione 4
4. Unità didattica 2: lezione 5
5. Unità didattica 2: lezione 6

ESERCITAZIONI DI LABORATORIO 2

IL SOFTWARE DOXYGEN

Cos'è Doxygen

La spiegazione sintetica ed esauriente di cosa sia ◀ **Doxygen** ▶ la si trova sul sito Internet dove esso viene distribuito gratuitamente (http://www.doxygen.org):

◀ **Doxygen** Doxygen is a documentation system for C++, C, Java, Objective-C, Python, IDL (Corba and Microsoft flavors), Fortran, VHDL, PHP, C#, and to some extent D.
It can help you in three ways:

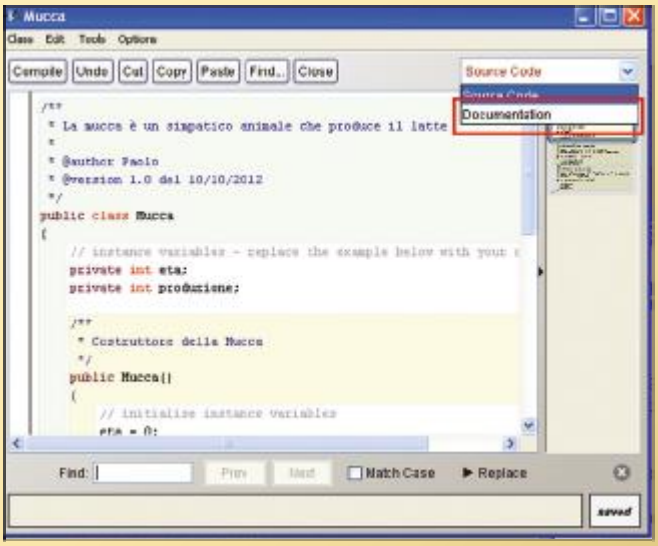
1. It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in LaTeX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.
2. You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. You can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.
3. You can also use doxygen for creating normal documentation.

Doxygen is developed under Linux and Mac OS X, but is set-up to be highly portable. As a result, it runs on most other Unix flavors as well. Furthermore, executables for Windows are available. ▶

Doxygen è un tool per la generazione automatica della documentazione di codice sorgente (C, Java ecc.) partendo dai singoli commenti presenti nel codice stesso, opportunamente formattati.
La formattazione avviene inserendo dei commenti nel codice per impaginare una documentazione utilizzando particolari tag.

I programmatori Java hanno già a disposizione uno strumento simile che produce la documentazione automatica: Javadoc, che da Bluej si genera direttamente selezionando l'opzione Documentation. ▶

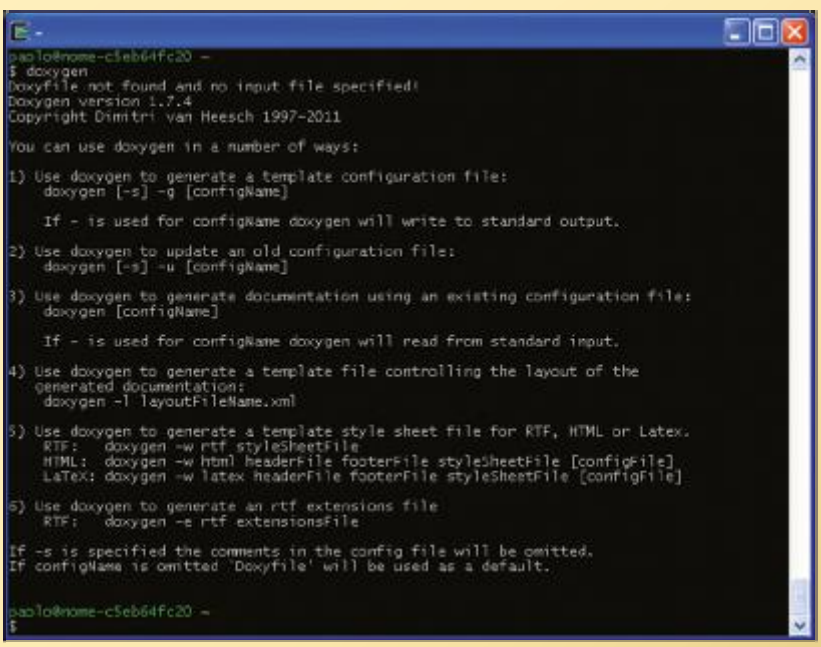
Doxygen è molto simile a Javadoc e utilizza una sintassi dei comandi praticamente uguale: anch'esso genera un ipertesto HTML ma aggiunge molte funzionalità tra cui la generazione dei file in formato ◀ Latex ▶ e la creazione, a partire da esso, di un documento: il Reference Manual.



◀ Latex Latex (scritto anche LaTeX) è un linguaggio di markup usato per la preparazione di testi basato sul programma di composizione tipografica TeX. ▶

Installare Doxygen

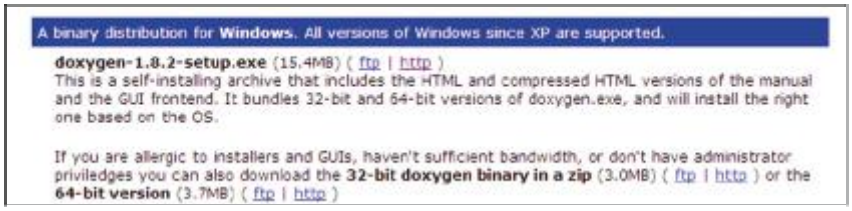
Se avete installato Cygwin non c'è bisogno di installare Doxygen in quanto è già presente nell'ambiente: per verificarlo basta digitare *doxygen* al prompt e si visualizzerà la seguente schermata:



In alternativa, per installare Doxygen è necessario utilizzare un computer connesso a Internet: nella home page del sito http:// www.doxygen.org si può individuare il link per effettuare il download sia del programma che del manuale.

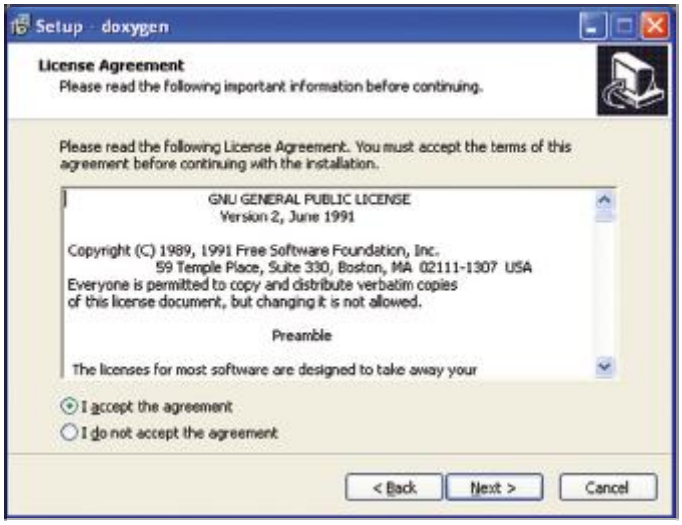
È anche possibile scaricare il programma dal sito www.hoepliscuola.it nella cartella materiali nella sezione riservata al presente volume.

Nel momento in cui viene scritta questa esercitazione, la versione di Doxygen più recente è quella comprendente doxygen-1.8.2-setup.exe. Le istruzioni che seguono faranno dunque riferimento all'installazione di questa specifica versione.



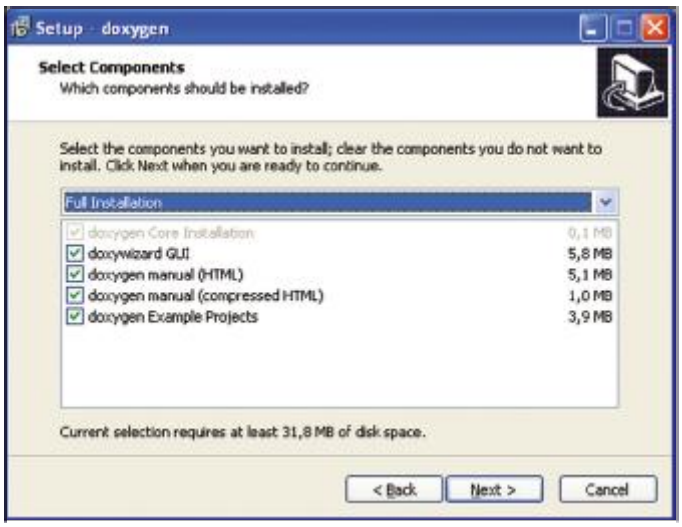
Un semplice click con il pulsante sinistro del mouse su html inizia il processo di installazione: come prima passo è necessario autorizzare l'esecuzione del programma di installazione cliccando Esegui nella finestra di protezione che ci si presenta davanti oppure Salva per scaricare una copia sul nostro hard disk. ▶

Noi salveremo una copia nel nostro PC, per esempio nella cartella hoepli, per poi avviare l'installazione, che proporrà la seguente schermata: ▶



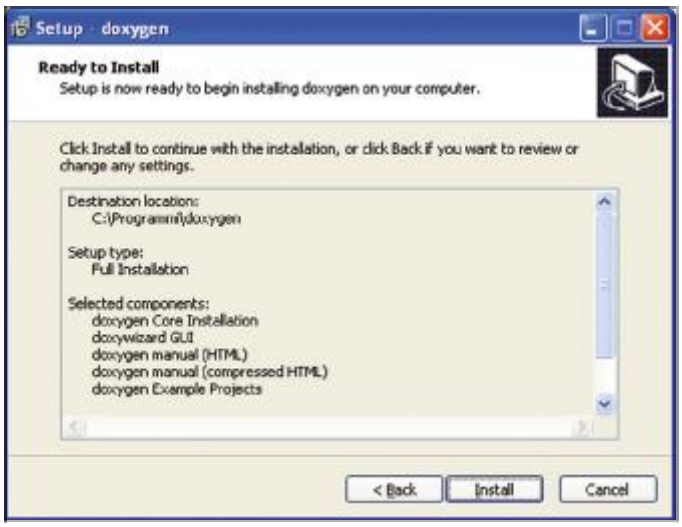
Accettando i termini della licenza d'uso si avvia l'installazione scegliendo la directory di destinazione: ▶

Lasciamo settati tutti i flag per effettuare l'istallazione completa di tutte le opzioni: ▶

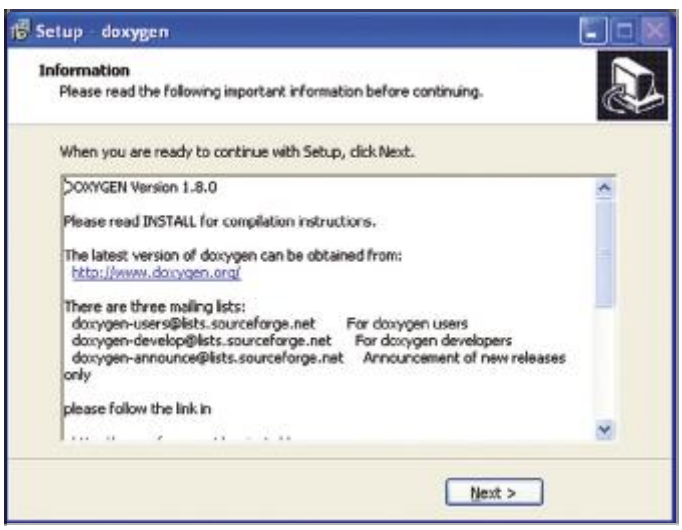


e creiamo una shorcut dal nostro menu delle applicazioni di window, utile per mandare successivamente in esecuzione il programma: ▶

ora siamo pronti per installare il programma: clicchiamo sul pulsante Install dalla successiva videata ▶



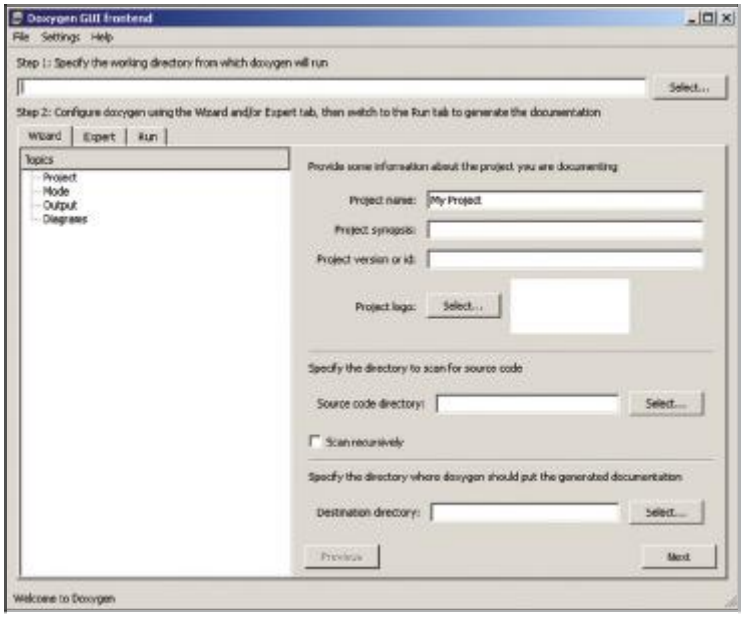
e dopo aver eletto le informazioni sulla versione completiamo l'installazione sempre tramite il pulsante Install. ▶



Al termine dell'istallazione possiamo mandare in esecuzione il programma dal menu programmi, selezionando il link Doxywizar. ▶

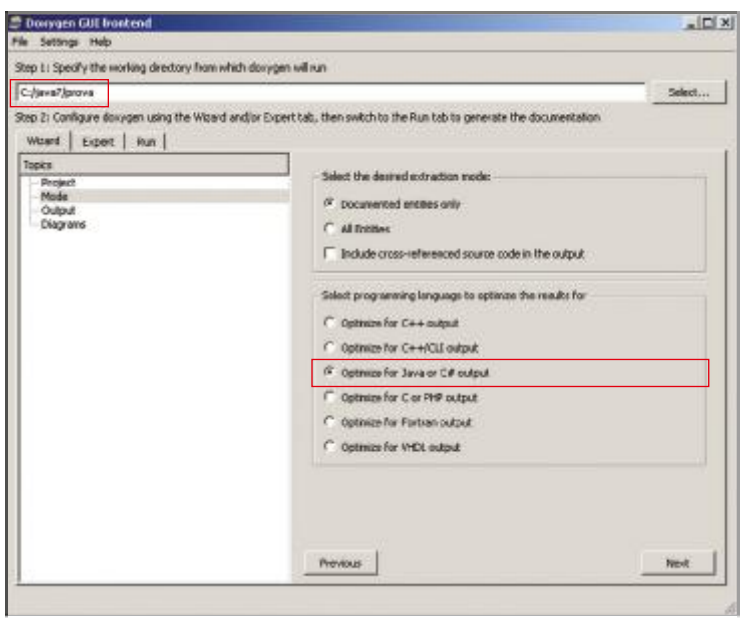
Generazione della documentazione da window

La videata iniziale è la seguente: ▶

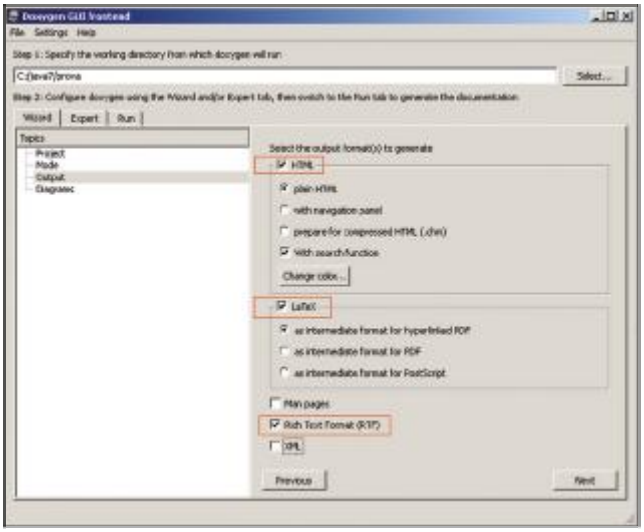


È possibile assegnare un nome al progetto e indicare le directory dove Doxygen deve essere mandato in esecuzione (cartella di riferimento, nel nostro esempio è la cartella *prova* dove è presente un singolo file Java, *Mucca.java*), dove sono presenti i codici sorgenti e dove deve essere memorizzata la documentazione prodotta.

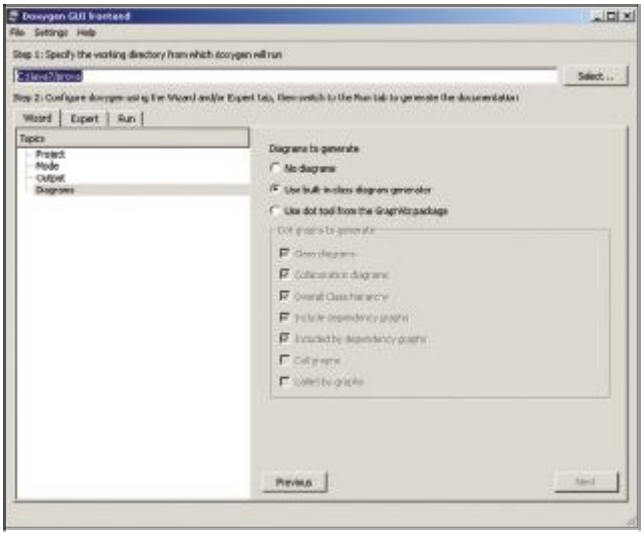
Cliccando su NEXT o selezionando mode nella finestra wizard è possibile scegliere il linguaggio di programmazione: ▶



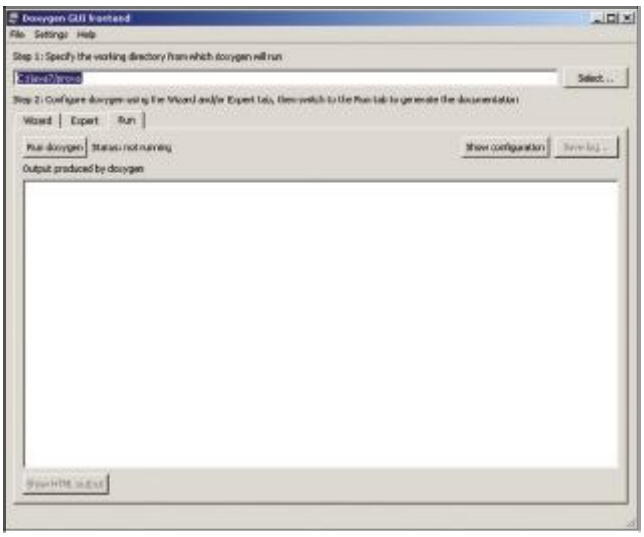
Nella successiva finestra si "spunta" il formato nel quale si desidera venga prodotta la documentazione: ▶



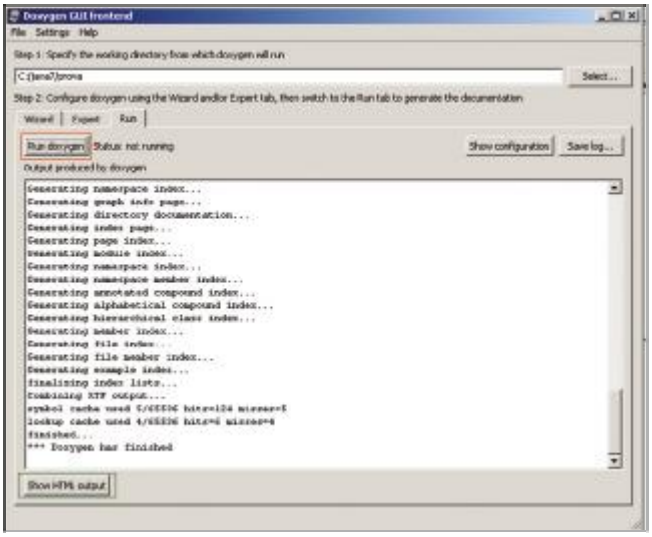
La successiva videata permette di indicare a Doxygen se devono essere generati anche dei diagrammi ▶



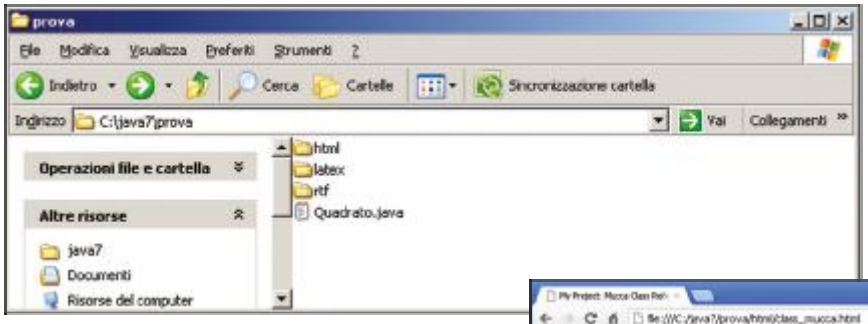
L'ultima tab, quella con titolo RUN, è la finestra di esecuzione dove viene avviata la generazione della documentazione e viene visualizzato l'echo dei comandi eseguiti dai singoli passi dell'evoluzione di Doxigen: clicchiamo il pulsante Run doxygen ▶



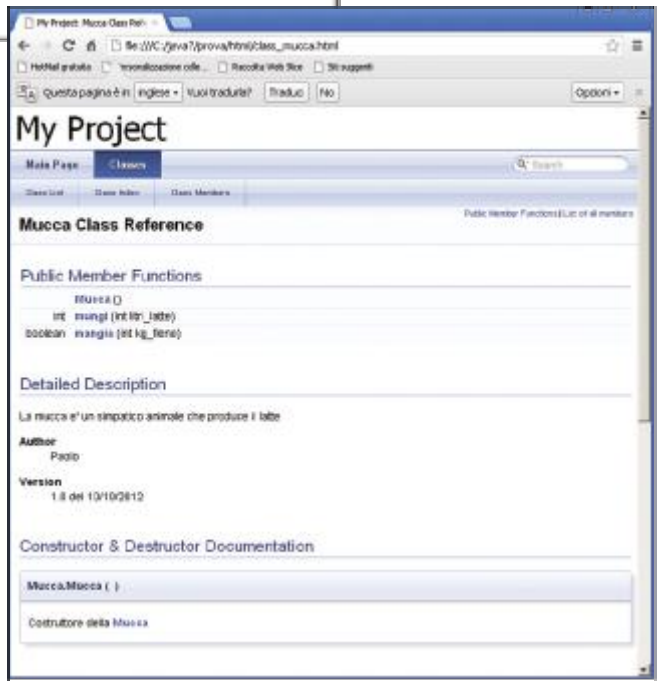
e otteniamo una esecuzione simile alla seguente ▶



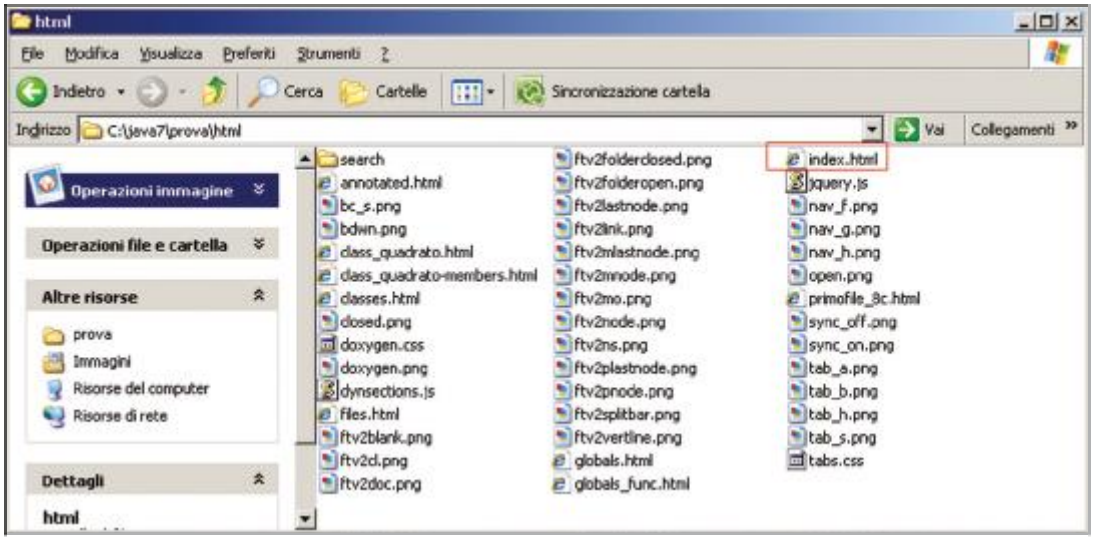
Al termine possiamo direttamente visualizzare l'output generato in formato HTML cliccando sull'apposito pulsante: se invece apriamo la cartella *prova* possiamo osservare che sono presenti tre nuove cartelle, una per ogni formato di documentazione che abbiamo precedentemente richiesto:



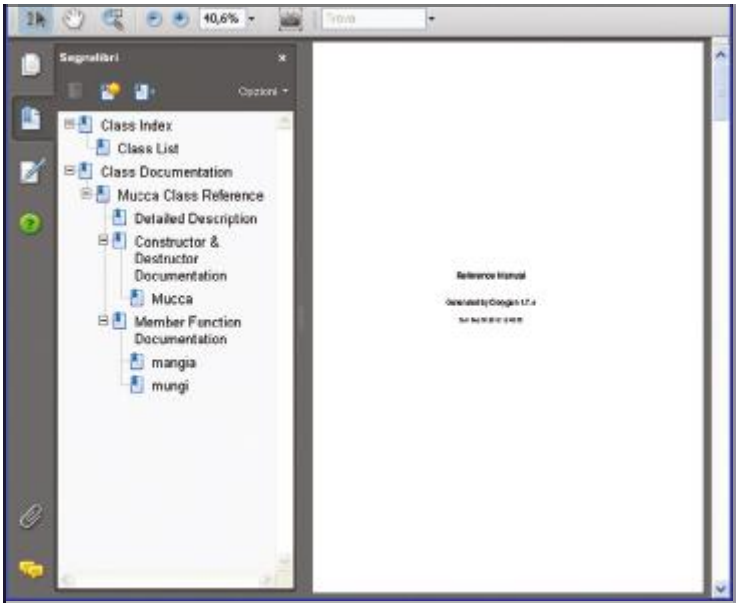
Visioniamo il codice HTML e nel browser ci viene proposto il seguente ipertesto (molto simile a quello prodotto da Javadoc): ▶



Se visioniamo il contenuto della cartella HTML troviamo tutti i file

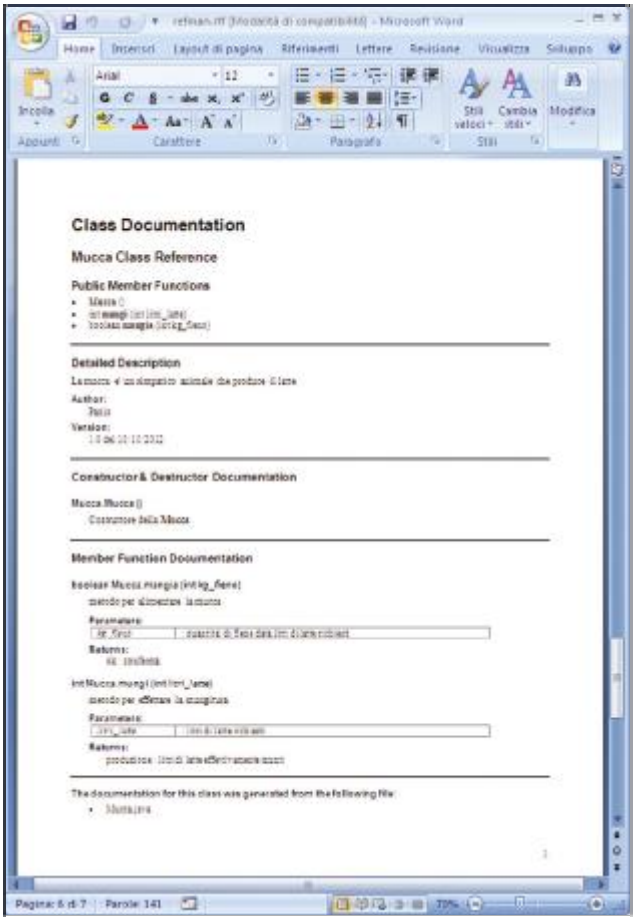


Entriamo nella sottodirectory latex e mandiamo in esecuzione il file make.bat: viene compilato automaticamente il manuale in PDF, come quello riportato di seguito. ▶



Come ultima operazione entriamo nella sottodirectory rtf

e troviamo la documentazione anche in forma rtf, leggibile ed editabile con MS Word. ▶



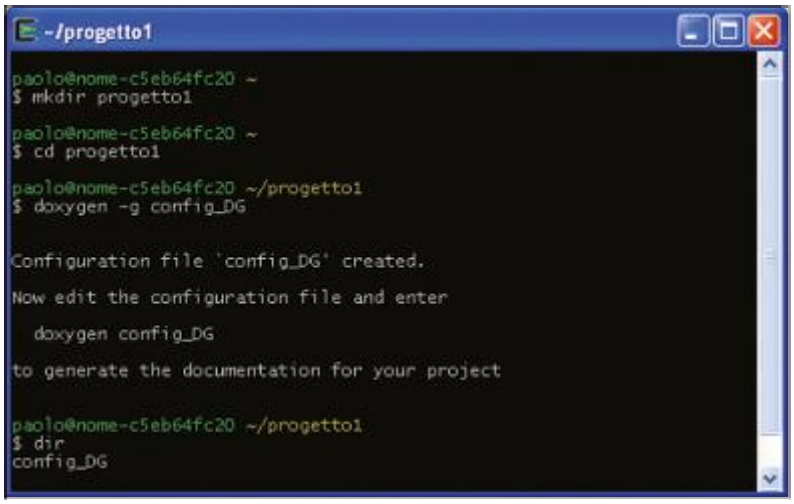
Generazione della documentazione da linux/cgwin

Creiamo un nuovo progetto e predisponiamo il file di configurazione di default con il comando

```
doxygen -g <nomefile>
```

dove sostituiamo <nomefile> col nome che vogliamo assegnare al nostro file, per esempio *config_DG*. ▶

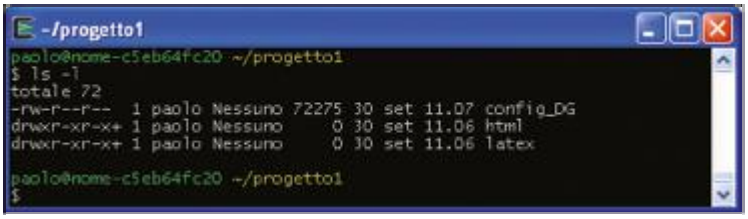
Il file config_DG può essere aperto in qualunque editor di codice (per esempio Dev-Cpp) oppure anche con MS Word.



Si prosegue editando il file di configurazione appena generato settando i seguenti parametri:

```
PROJECT_NAME      = nome del progetto
OUTPUT_DIRECTORY  = cartella in cui generare la documentazione
INPUT             = percorso del codice  sorgente
FILE_PATTERNS     = Estensioni da considerare, per esempio .h,.cpp, .cc ecc...
```

La documentazione verrà generata in formato html nella cartella "html". ▶



Per generare la documentazione automatica in qualsiasi momento è sufficiente digitare

```
doxygen <nomefile>
```

Prima di generare la documentazione è necessario inserire nel codice la descrizione di classi, metodi, e attributi, come descritto in seguito.

Elenco dei comandi

I commenti possono essere:

// *Commento su linea singola*

/*
* *commento su più linee*
*/

I commenti utilizzano alcuni particolari tag di formattazione che, uniti a particolari parole riservate, permettono di far risaltare nella documentazione diverse informazioni, tipo il nome della funzione, l'autore, la data di creazione o modifica, i parametri, il file sorgente di appartenenza o più semplicemente un breve commento o nota.
Vediamo ora alcuni tipi di tag, che devono essere scritti compresi tra

/*
< riga iniziante con TAG per essere visualizzata nella documentazione>
*/

Tag	Descrizione	Esempio	
\fn	serve per identificare il nome della funzione	\fn	int somma(int a, int b)
\brief	permette di visualizzare un commento	\brief	funz. che somma i valori interi a e b
\param	permette di identificare sulla documentazione un parametro di funzione e di applicarvi una descrizione	\param \param	int a: Primo valore da sommare int b: Secondo valore da sommare
\return	permette di descrivere ciò che ritorna la funzione	\return	int: la funzione ritorna la somma int dei due parametri
\date	permette di visualizzare sulla documentazione una data, per esempio quella di creazione o modifica	\date	12/10/2010

Tag	Descrizione	Esempio
\author	permette di stampare il nome dell'autore dell'ultima modifica o chi ha creato la funzione	\author Zio Pino
\file	permette di visualizzare sulla documentazione il file sorgente su cui si trova la funzione in questione	\file primoesempio.c
\version	permette di visualizzare sulla documentazione la versione del file	\version 4.1
\bug	permette di visualizzare sulla documentazione l'ultimo bug trovato	\bug problema corretto Out of Range
\class	permette di visualizzare sulla documentazione il nome della classe che si commenta	\class <nome della classe>
\warning	permette di visualizzare sulla documentazione l'ultimo bug trovato	\warning la funzione ritorna int, ma la somma può dare un double
\example	permette di specificare un file di esempio. Nell'html verrà richiamato un file sorgente di esempio	\example nomefile_esempio.c

Per realizzare una documentazione semplice ma completa è sufficiente avere l'accortezza di inserire i tag di commento almeno in quattro posizioni:

1. all'inizio di ogni file inserire le seguenti righe:

```
/*
   @file        fileName.cc .cpp .h ecc ecc ecc
   @author      name, mail
   @version 1.0
   */
```

2. prima della dichiarazione di una classe:

```
/** Descrizione classe...
   @code
   ... eventuale codice di esempio ...
   @endcode
   */
```

3. prima di ogni metodo (o ridefinizione o template):

```
/**
   Descrizione metodo...
   @param      a parametro 1
   @param      b parametro 2
   @return     valore di ritorno
   @throws     eccezioni...
*/
```

4. prima di ogni attributo:

```
/** descrizione */
```

In questo modo si otterrà una documentazione completa ed esaustiva, come quella riportata nei successivi esempi.

ESEMPIO 12 *Documentiamo una classe C*

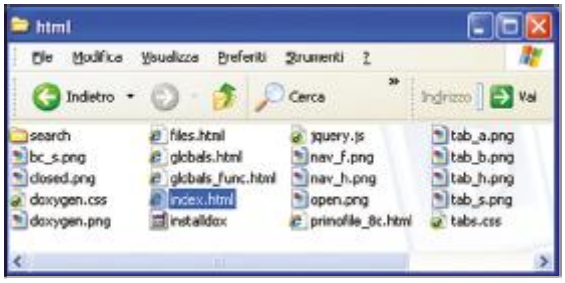
Come primo esempio creiamo la documentazione per un semplice file C: creiamo una directory progettoC e scriviamo il seguente programma salvandolo nel file primofile.C ▶

```
1  //  \file      primofile.c
2  //  \brief     file di esempio
3  //  \date      il file è stato creato il 10/10/2012
4
5  /**
6  * \fn          int somma(int a, int b)
7  * \brief       funzione che somma i parametri a e b
8  * \param       int a: Primo valore da sommare
9  * \param       int b: Secondo valore da sommare
10 * \return      int: la funzione ritorna la somma int dei due parametri
11 * \date        12/10/2012
12 * \author      Zio Pino
13 * \file        primofile.c
14 */
15 int somma(int a, int b)
16 {
17  return a + b ;
18 }
```

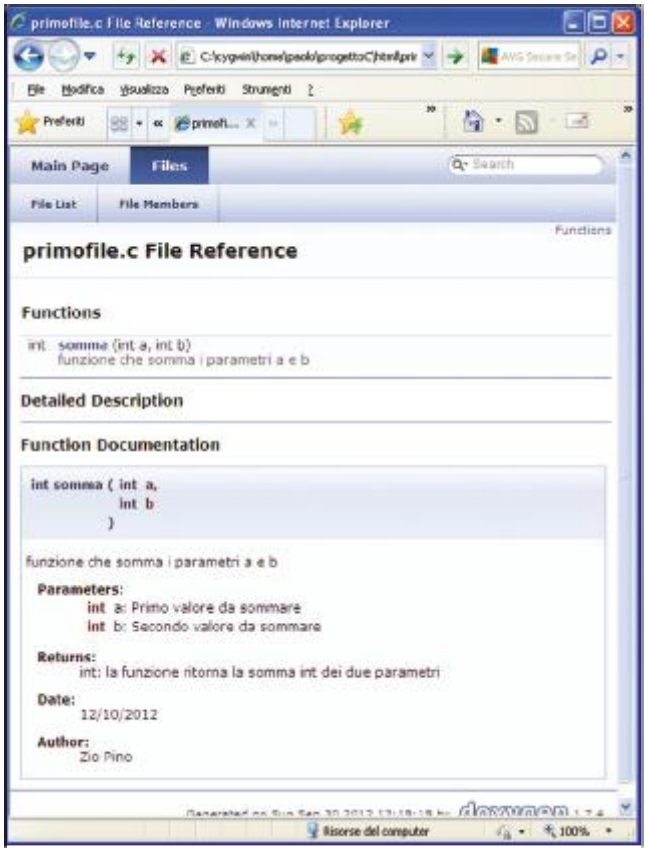
Digitando il comando

```
doxygen config_DG
```

si genera la documentazione sia in formato html che Latex: esplorando la cartella html ▶



selezioniamo index.html per visualizzarlo in un browser, come riportato di seguito: ▶



ESEMPIO 13 *Documentiamo una classe Java*

Come secondo esempio generiamo la documentazione per la classe Mucca.java di seguito riportata:

```
/**
 * La mucca e' un simpatico animale che produce il latte
 *
 * @author Paolo
 * @version 1.0 del 10/10/2012
 */
public class Mucca
{
    // instance variables - replace the example below with your own
    private int eta;
    private int produzione;

    /**
     * Costruttore della Mucca
     */
    public Mucca()
    {
        // initialise instance variables
        eta = 5;
    }

    /**
     * metodo per effettuare la mungitura
     *
     * @param  litri_latte : litri di latte richiesti
     * @return produzione  : litri di latte effettivamente munti
     */
    public int mungi(int litri_latte)
    {
        // put your code here
        return produzione;
    }
    /**
     * metodo per alimentare la mucca
     *
     * @param  kg_fieno : quantita' di fieno data litri di latte richiesti
     * @return ok       : conferma
     */
    public boolean mangia (int kg_fieno)
    {
        boolean ok;
        ok=true;
        return ok;
    }
}
```

viene prodotta la seguente documentazione ▶

Mucca Class Reference - Windows Internet Explorer

Main Page | Classes

Class List | Class Index | Class Members

Public Member Functions

Mucca Class Reference

List of all members.

Public Member Functions

	Mucca ()
int	mungi (int litri_latte)
boolean	mangia (int kg_fieno)

Detailed Description

La mucca e' un simpatico animale che produce il latte

Author:
Paolo

Version:
1.0 del 10/10/2012

Constructor & Destructor Documentation

Mucca::Mucca ( ) [inline]

Costruttore della Mucca

Member Function Documentation

boolean Mucca::mangia (int kg_fieno) [inline]

GraphViz

GraphViz è un tool che deve esser scaricato a parte e che può essere utilizzato separatamente o integrato al Doxygen.
In breve ci permette di creare un grafo che mostra dove una funzione viene chiamata e da chi oppure che cosa chiama al suo interno.

Prova adesso!

1. Scarica il file Mucca.java, aggiungi almeno altri tre metodi e copiala in una nuova directory di cgwin\home\tuonome chiamandola

   progettoJava

2. lancia

   doxygen –g documenta

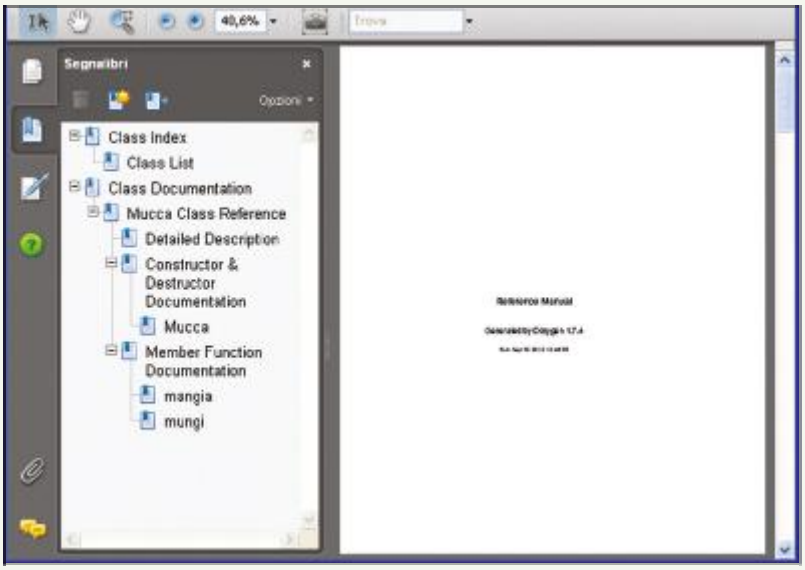
3. quindi digita il comando

   doxygen documenta

4. entra nelle sottodirectory create e osserva (ls) i vari output generati

5. visualizza con un browser il file index.html

6. entra nella sottodirectory latex e digita semplicemente make: viene compilato automaticamente il manuale in PDF, come quello riportato di seguito.



7. Per comprendere le potenzialità di Doxygen, guarda i seguenti esempi di documentazione:
   http://api.kde.org/
   http://www.opensg.org/doxygen/
   http://www.abisource.com/doxygen/index.html

Esercitazioni di Laboratorio 3

IL CONTROLLO DELLE VERSIONI CON SUBVERSION E TORTOISESVN

Premessa

Lo sviluppo del software è un processo lungo, generalmente fatto in squadra, che non termina con la consegna del prodotto: la manutenzione, sia adattativa che migliorativa, segue il prodotto sino alla sua dismissione o sostituzione con nuove soluzioni alternative.
Raramente un segmento di codice rimane invariato nel tempo, sia per le eventuali modifiche fatte in fase di sviluppo, che quelle successive al suo rilascio.
È necessario introdurre un sistema di identificazione del codice in modo che sia sempre possibile conoscere le sue funzionalità e l'evoluzione che le stesse hanno avuto nel tempo.

Spesso si utilizza un sistema di numerazione mediante tre contatori su tre livelli che vengono incrementati a partire da **1.0.0**: major level, minor level e patch level (per esempio: 2.14.11), dove:

- major level (release): indica una modifica sostanziale nel prodotto che può implicare modifiche nell'architettura, per esempio una nuova veste grafica oppure il passaggio a una nuova piattaforma;
- minor level (versione): si migliorano le funzionalità esistenti oppure si aggiungono nuove funzionalità alla versione precedente e/o aggiornamenti di natura fiscale;
- patch level (correzione): si rimuovono errori/malfunzionamenti.

> Non esiste comunque una regola standard, e quindi vi sono molti metodi differenti per la gestione dei numeri di versione e, di fatto, è sufficiente tenere a mente due regole essenziali:
> 1. ogni programmatore deve assegnare un numero a ogni nuova versione del programma;
> 2. a ogni evoluzione del software il numero deve essere maggiore del precedente.

Controllo versioni nello sviluppo con Subversion

Durante la fase di sviluppo viene aggiornato solamente un contatore in modo da indicare agli altri sviluppatori che utilizzano il nostro codice che ne è stata prodotta un versione migliorata.

> Può anche succedere che più sviluppatori stiano portando modifiche allo stesso programma, in sezioni diverse, e il mancato coordinamento sullo scambio reciproco dei file può portare alla perdita del lavoro di uno e dell'altro sviluppatore.

Per effettuare il controllo delle versioni esistono prodotti appositamente realizzati, e tra tutti Subversion è di fatto il nuovo standard, il più utilizzato sia perché gratuito (Open Source) sia perché offre un set di funzionalità e performance davvero rivoluzionarie.

È possibile visitare il sito ufficiale di Subversion all'indirizzo http://subversion.apache.org/ oppure leggere ulteriori informazioni discussioni, eventi, materiale didattico, esperienze dal sito di Subversion in Italia, all'indirizzo www.subversionitalia.it/.

Subversion non ha una interfaccia grafica, ma i comandi vengono dati via shell: sono però disponibili molte interfacce grafiche di supporto e per Windows la più utilizzata è ◀ **TortoiseSVN** ▶ che integrandosi in Explorer fornirà immediatamente i comandi attraverso l'interfaccia di Windows.

◀ **TortoiseSVN** TortoiseSVN is a Subversion (SVN) client, implemented as a windows shell extension. It's intuitive and easy to use, since it doesn't require the Subversion command line client to run. Simply the coolest Interface to (Sub)Version Control! ▶

Installazione di SubVersion e TortoiseSVN

Scaricate il software Subversion all'indirizzo http://www.subversionitalia.it/:

Attualmente l'ultima versione del software è la *Setup-Subversion-1.7.6.msi*.
Non ci sono particolari configurazioni da settare durante l'installazione di Subversion, che avviene semplicemente cliccando sull'icona come un qualsiasi altro programma.

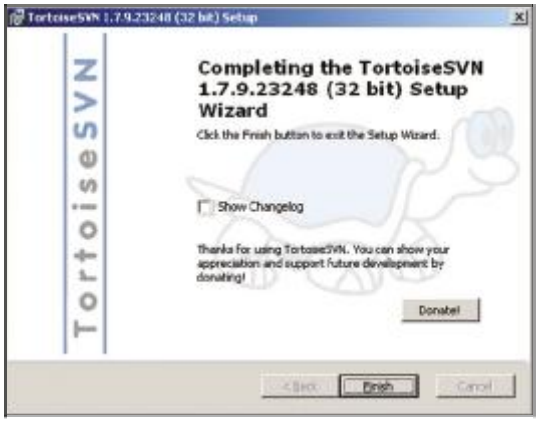
Ora procediamo al download di TortoiseSVN da http://tortoisesvn.net/downloads dove è possibile scegliere la versione da scaricare in base alla architettura del nostro pc (versione a 32 o 64 bit).
Attualmente l'ultima versione del software è *TortoiseSVN-1.7.10.23359-win32-svn-1.7.7.msi*.

Anche l'installazione di TortoiseSVN non presenta difficoltà: riportiamo semplicemente in sequenza le videate che si presenteranno alle quali è sufficiente confermare ogni operazione cliccando sul pulsante Next. ▶

È possibile trovare questi file nella cartella materiali nella sezione del sito www.hoepliscuola.it riservata a questo volume.

Si confermano successivamente tutte le successive videate, dopo aver accettato i termini della licenza d'uso:

Si avvia quindi l'installazione, che termina rapidamente con la seguente schermata: ▶



Al termine dell'installazione viene richiesto di riavviare il computer per terminare la configurazione inserendo i link nel menu dei comandi, come mostrato nella seguente figura:

È necessario riavviare il computer anche per attivare l'Icon Overlay dato che TortoiseSVN, essendo un plugin della shell di Windows, viene caricato contestualmente a quest'ultima.

Uso di Subversion su un nostro progetto

Descriviamo il funzionamento di Subversion direttamente mediante un esempio.

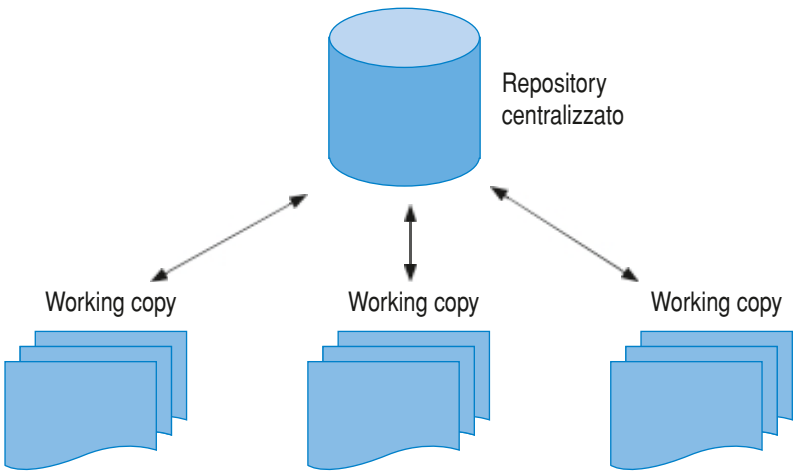
Riportiamo prima la spiegazione della terminologia utilizzata da Subversion:

- Repository: con repository si intende l'archivio centralizzato di dati gestito direttamente da Subversion, cioè la copia centralizzata dove tutti gli sviluppatori inviano i loro file con una operazione di commit e dalla quale scaricano i file aggiornati dagli altri programmatori mediante una operazione di update.

Il repository deve contenere le ultime versioni dei file, quelle con gli ultimi aggiornamenti.

- Working Copy: è la copia personale dei file sulla quale lavora ciascun programmatore, cioè è la propria copia di lavoro dove vengono apportate le modifiche e quindi inviati i file al repository non appena si termina una rettifica/integrazione.
- Revisione: ogni file che viene spedito al repository in sostituzione di uno precedente è chiamato revisione e viene numerato automaticamente incrementando il contatore da 1 in avanti.
- Commit: è l'operazione che trasmette le nuove copie dal Working Copy al Repository;
- Update: è l'operazione che scarica le nuove versioni dal Repository al Working Copy.

Se per esempio abbiamo tre utenti che lavorano nel progetto ci troviamo nella seguente situazione:

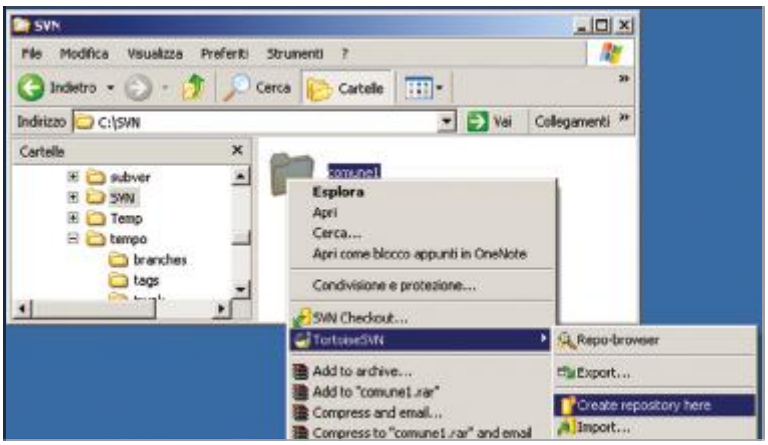


Creazione del repository

1. Creiamo la cartella destinata a condividere i file del nostro progetto, per esempio:

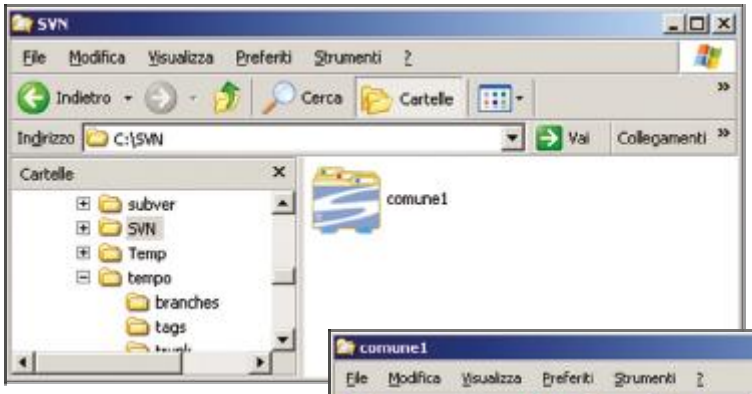
   C:\SVN\comune1

2. Rendiamola ufficialmente repository del nostro progetto, cliccando col tasto destro sulla cartella e selezionando TortoiseSVN/Create repository here…

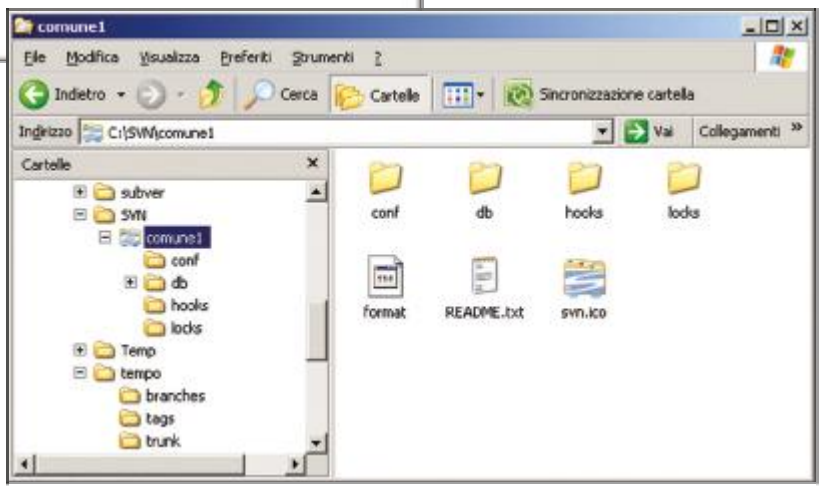


Creiamo prima la struttura con Create folder structure ▶

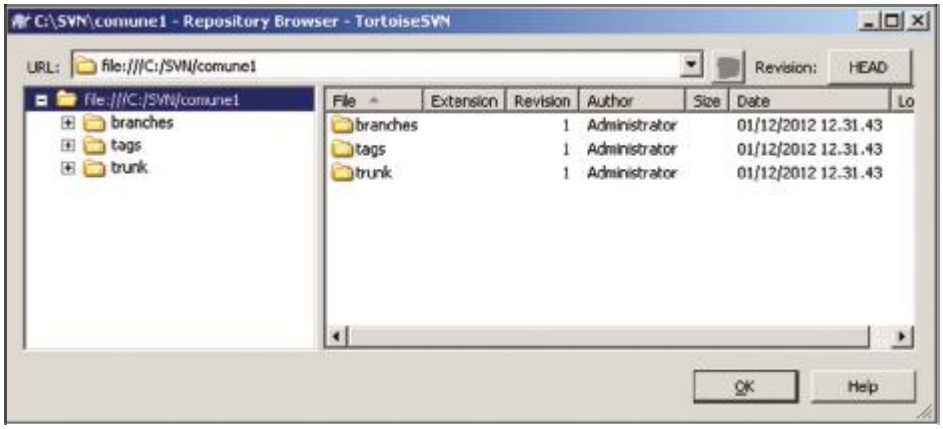
quindi viene cambiata l'icona sulla cartella:



Posizioniamoci sulla cartella: se tutto è andato a buon fine sono state create alcune directory, come si può vedere dalla figura a fianco: ▶

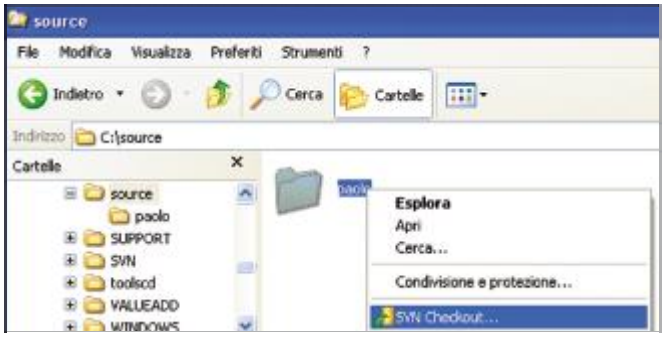


e cliccando su Start repobrowser viene visualizzato lo stato del repository:

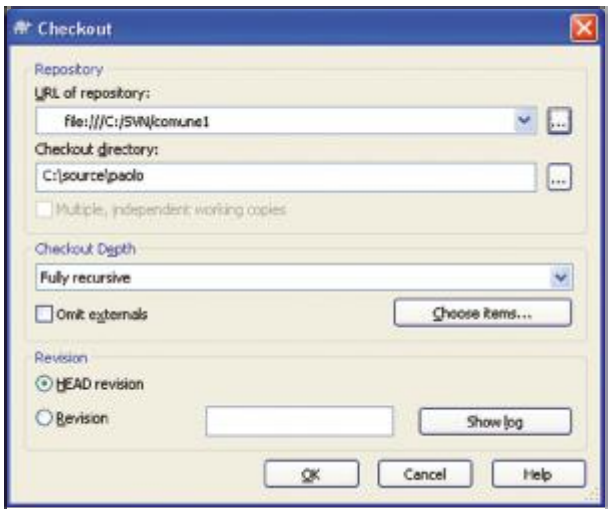


Inizializzazione work area: checkout

❸ Per poter iniziare a condividere i file è necessario "dichiarare" a SVN quale sarà la nostra work area, e questo viene effettuato mediante l'operazione di checkout.
Effettuiamo la creazione della directory di lavoro, per esempio c:\source\paolo.
Quindi ci posizioniamo su di essa e visualizziamo col tasto destro il menu:



Configuriamo la nostra working area "collegandola" al repository mediante la funzione SVN checkout:



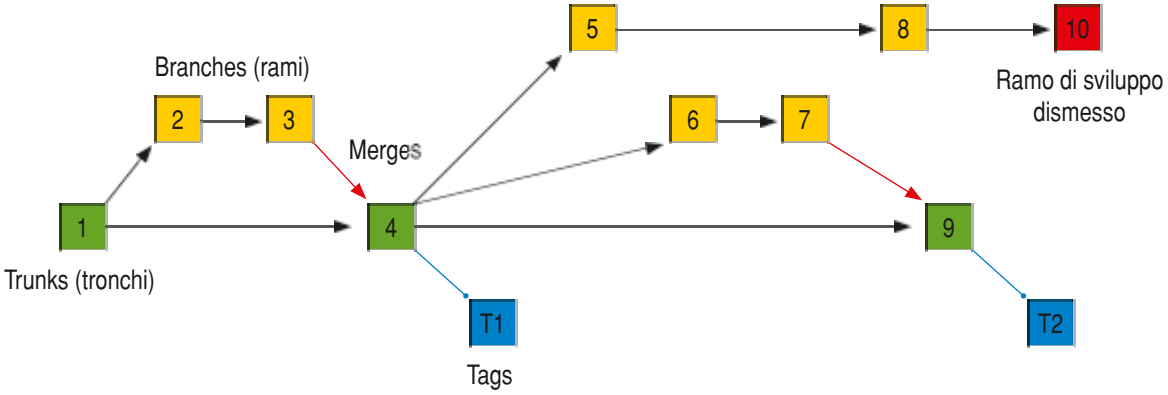
confermando con OK si ottiene l'importazione del repository nella nostra area di lavoro:

Nella nostra cartella di lavoro Working Copy sono ora presenti alcune directory:

- trunk è la cartella principale del nostro progetto, al suo interno verranno salvati i file del progetto in corso condivisi/da condividere con tutti gli sviluppatori;
- in branches ci saranno le eventuali ramificazioni del nostro progetto in versioni parallele di sviluppo;
- tags serve per funzioni avanzate per rendere organizzato il lavoro allo sviluppatore;
- .svn; è una cartella di servizio utilizzata da SVN.

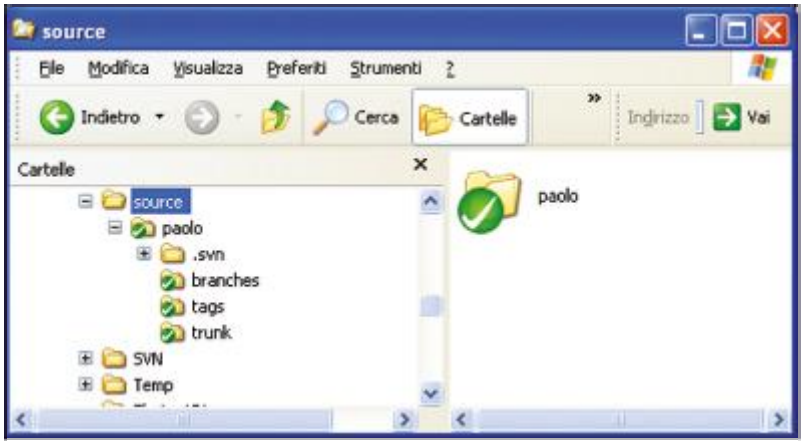
ESEMPIO 14

Un esempio di evoluzione del progetto potrebbe essere quella indicata nel seguente schema:



Possiamo vedere come il progetto ha due nodi particolari oltre al trunk, quelli indicati in verde, che sono punti di merges, dove cioè sviluppi paralleli di codice fatti da gruppi diversi di programmatori devono "sincronizzarsi" per procedere con una nuova fase di sviluppo che può essere, come in questo caso, ancora suddivisa in sezioni parallele.

Tutte hanno una nuova icona, che indica la situazione corrente dei file (in questo caso aggiornati col repository):



La spunta verde significa che il contenuto della cartella è aggiornato con quello del repository.

Prima importazione del nostro progetto

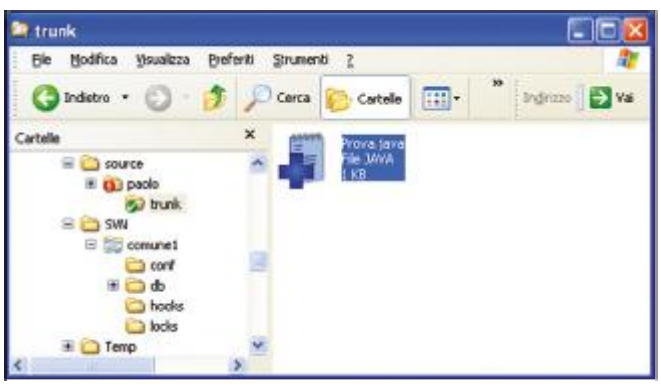
Proseguiamo ora effettuando la prima importazione nel repository del nostro progetto: per prima cosa inseriamo un file nella cartella trunk della nostra working area: il file viene marcato con l'icona di figura:

per indicare che è ancora in "lavorazione" in locale.

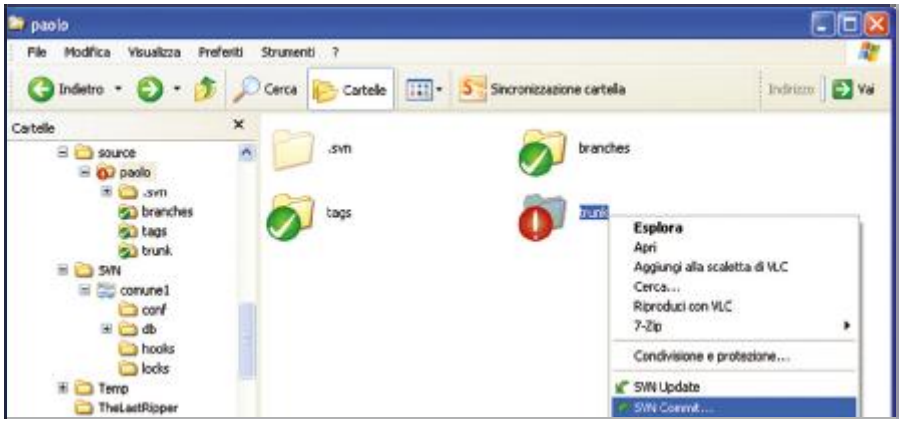
Per aggiungere un nuovo file al progetto e quindi al repository è necessario fare due operazioni:

A lo si clicca sempre con il tasto destro, si seleziona TortoiseSVN e successivamente Add… per "marchiarlo" in modo da selezionarlo per poi effettuare l'esportazione, confermando l'operazione con ok;

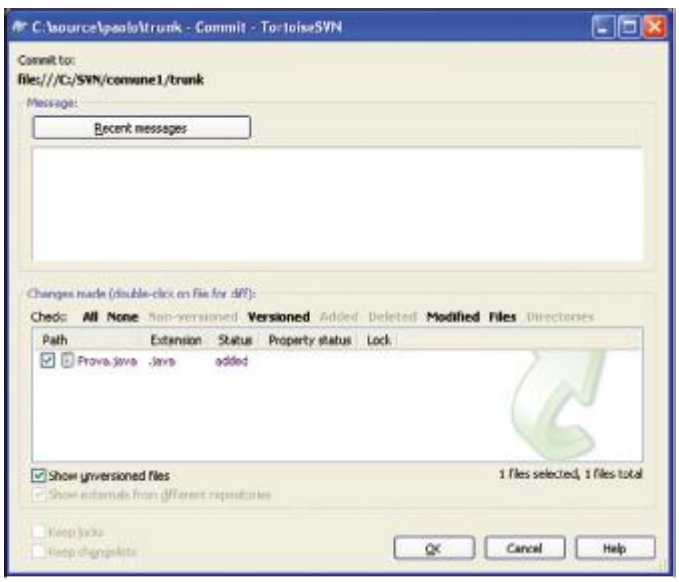
Dopo la conferma, il file selezionato ha una nuova icona:



B per aggiungerlo al repository è necessaria una fase di commit, come quella descritta in precedenza: sempre cliccando col tasto destro si procede a selezionare TortoiseSVN e successivamente Commit … il file viene aggiunto e incrementato il numero di versione.

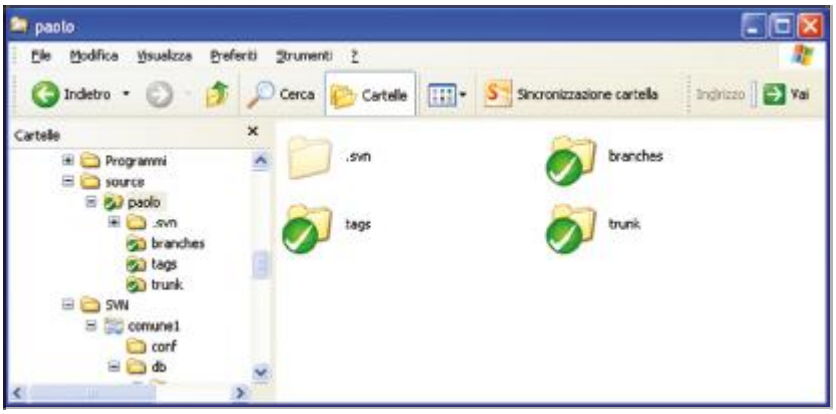


Nella nuova finestra ci viene riportata l'operazione che stiamo effettuando richiedendoci la conferma, che noi effettuiamo cliccando sul bottone ok. ▶



La nuova versione del progetto è stata creata, e le è stato assegnato il numero 2.

SVN ha modificato le icone della nostra working area, in modo che visivamente possiamo verificare che l'operazione è andata a buon fine.

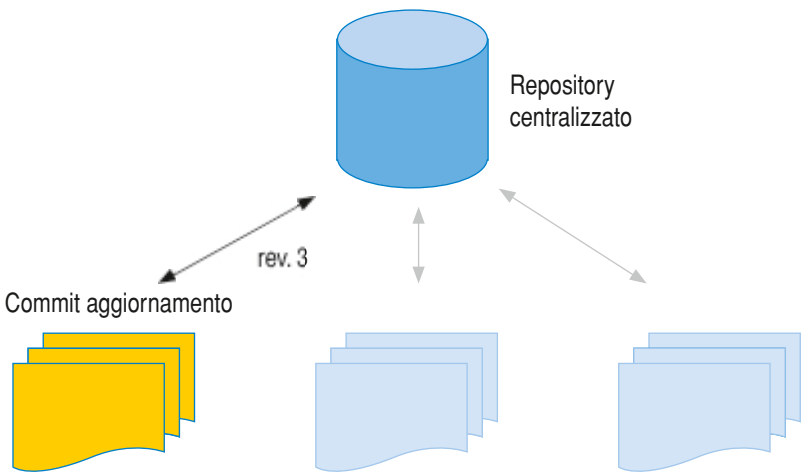


Modifica di un file e commit

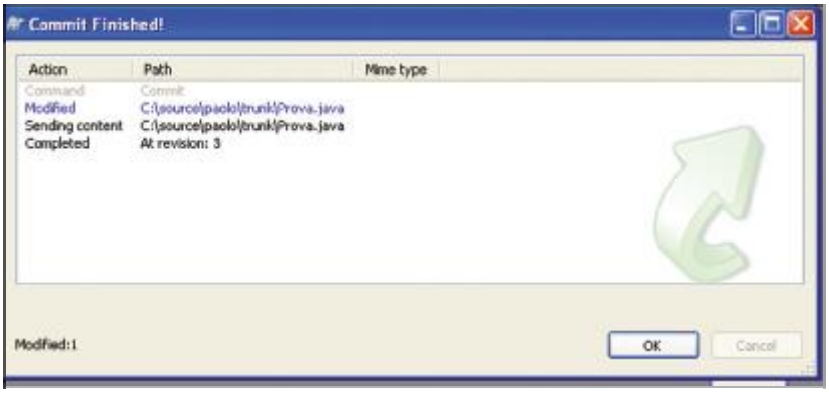
Posizioniamoci ora nella nostra work area e modifichiamo un file nella cartella trunk.

Al termine delle modifiche le icone vengono cambiate per ricordarci che è necessario fare il commit sul repository per condividere la nuova versione: ▶

L'aggiornamento lo effettuiamo con una operazione di commit:



lo trasferiamo sul repository semplicemente cliccando col tasto destro sul file da trasferire, selezionando TortoiseSVN e cliccando su commit

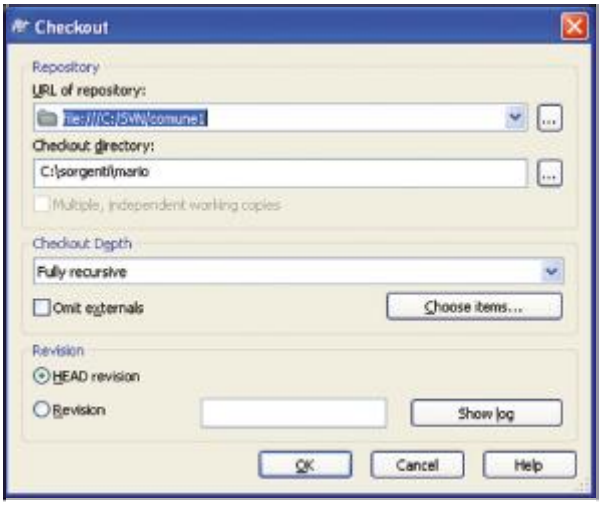


Se **erroneamente** clicchiamo su update verrà scaricato nella nostra directory il file presente sul repository, che è quello precedente le modifiche, che si sovrapporrà al nostro file facendo perdere tutto il nostro lavoro!

Osserviamo come dopo il commit SVN aggiorna il numero di versione.

Aggiunta di un nuovo programmatore

Supponiamo che un nuovo programmatore (mario) si aggiunga al gruppo: nella directory sorgenti/mario scarichiamo la copia del repository con checkout: ▶



Ora mario aggiunge un nuovo file e modifica quello presente: la sua cartella si presenta con questa strutturazione ▶

Aggiorniamo ora il repository con commit:

SVN riconosce che un file è stato aggiornato e quindi provvederà a incrementare il suo numero di versione: individua inoltre un nuovo file, non ancora "versionato" al quale assegnerà lo stesso numero di versione degli altri file.

> Il numero di versione non è riferito al singolo file, ma al **progetto**: ogni rettifica incrementa il contatore generale e aggiorna con quel numero tutti i file modificati o inseriti nella stessa operazione di commit.

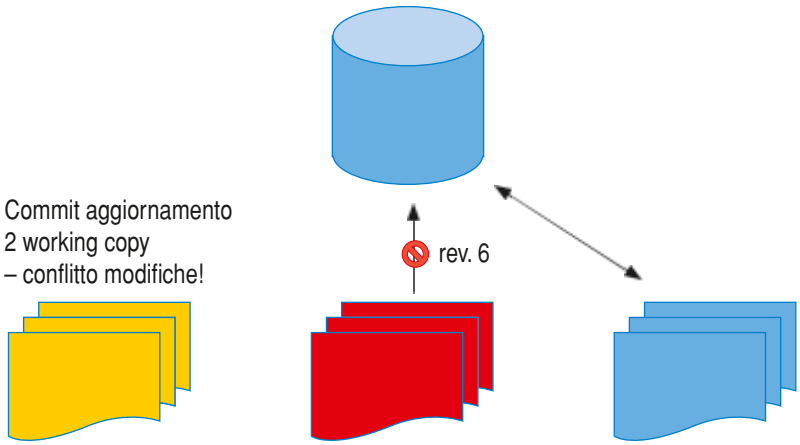
Settiamo il **check box** affinché il file venga aggiunto e confermiamo con ok.

Conflitto di aggiornamento

Supponiamo ora che il primo programmatore faccia anch'esso una modifica al programma Prova.java ed effettui il commit: questa operazione non va a buon fine e viene segnalato che sul repository è presente una versione di quel programma più recente

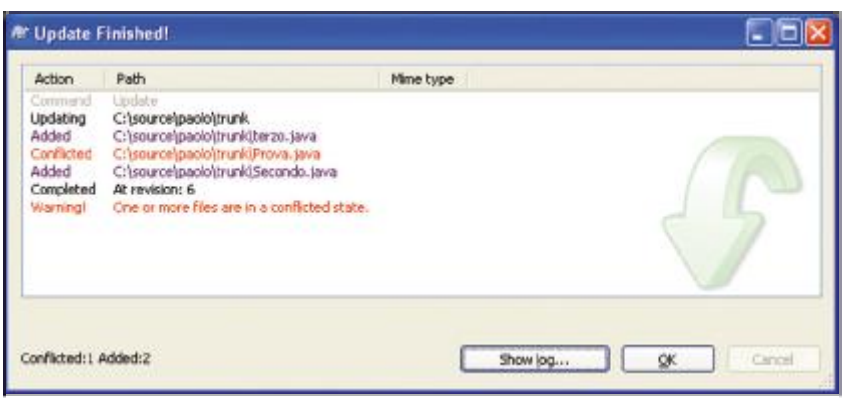
e invita a effettuare l'update prima di poter proseguire con il commit. ▶

Siamo in questa situazione:



Prima di iniziare a lavorare è quindi sempre opportuno sincronizzare la propria a working area con il repository, in modo da lavorare sempre sull'ultima versione del progetto.

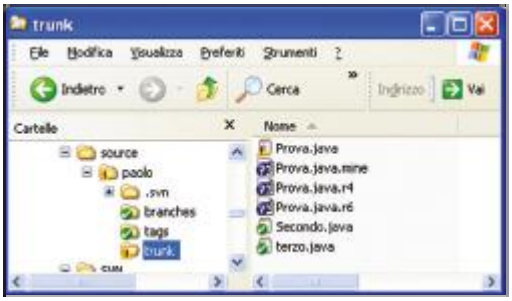
Confermando l'update, SVN prima di trasferire i file è bene controllare quelli che sono nella nostra working area e scaricare solo quelli presenti nel repository che sono più aggiornati di quelli locali: per quanto riguarda il file Prova.java ci segnala che sul repository è presente una versione diversa da quella locale, ma la versione locale non rispetta la sequenza di versione, quindi entrambe contengono rettifiche differenti fatte da programmatori diversi in contemporanea.



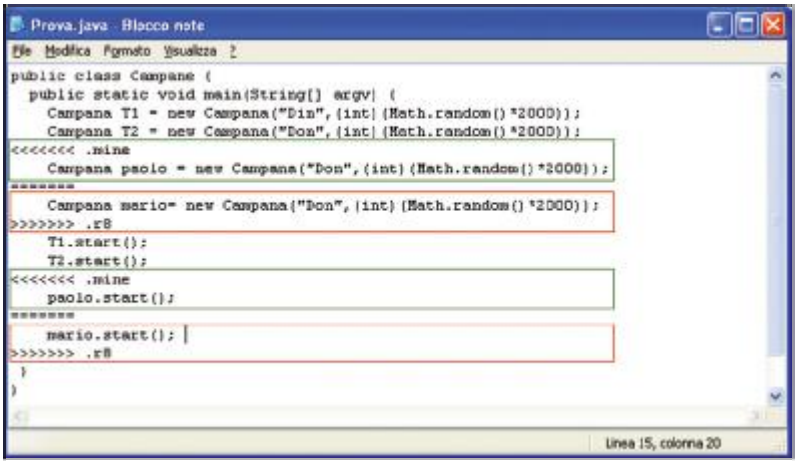
Cliccando sul bottone show log possiamo vedere la "storia" del nostro file

e scopriamo che per ultimo mario ha aggiunto una rettifica (che noi non abbiamo scaricato)!

Se ora apriamo il nostro direttorio troviamo la seguente situazione: ▶



SVN ha rinominato il nostro file il Prova.java.mine, ha scaricato la r4 (versione precedente a tutte le rettifiche), la r6 (ultima versione), ha generato da solo un file Prova.java dove ha "mixato" le rettifiche, cioè ha aggiunto alla r6 di mario le nostre correzioni, proponendoci il file finale dove sono evidenziate tutte le rettifiche apportate da entrambi i programmatori:



In rosso abbiamo indicato le rettifiche che sono state apportate alla versione r6 da parte di mario e in verde le nuove rettifiche che ha appena effettuato paolo e non sono ancora presenti nel repository: per procedere, è sufficiente cancellare tutti i file di servizio (.r4, .r6, .mine) ed effettuare il commit.

Conclusioni

Obiettivo di questa esercitazione non era quello di fornire la guida definitiva a Subversion, bensì descriverne a grandi linee il funzionamento. Infatti sono molteplici le possibilità offerte e solo utilizzandolo se ne comprendono appieno le potenzialità.

Esiste anche una versione di Subversion client/server per Internet, che si fonda sul seguente concetto: il repository è uno unico, gestito da una installazione di Subversion Server e ha il compito di organizzare il salvataggio e la gestione completa delle revisioni.
Ogni utente dispone invece di una versione di Subversion Client, che è l'interfaccia al repository: questo interpreta tutti i comandi necessari per scaricare una revisione, inviare gli aggiornamenti e analizzare le modifiche.

> È necessario disporre di un server che supporti SVN per evitare l'installazione e la configurazione della versione server: a tal scopo valutate cosa offre il vostro provider (per esempio Dreamhost lo integra nei propri piani di hosting).

NOTE

e-ISBN 978-88-203-5821-1

www.hoepli.it

Ulrico Hoepli Editore S.p.A.
via Hoepli, 5 - 20121 Milano
e-mail hoepli@hoepli.it