

# Il modello a 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 diventando un **processo**; evolve man mano che le istruzioni vengono eseguite dalla **CPU** e diviene quindi un'**entità attiva**: il **processo** è l'**istanza** di un **programma in esecuzione**.

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



### PROCESSO

Un **processo** è un'**entità logica (programma)** 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 scheduling (di **scheduling**) per determinare quando interrompere l'evoluzione di un processo e servirne un altro.

Questa tecnica di gestione della **CPU** si chiama **multiprogrammazione** e 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.

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.

### ESEMPIO

Si pensi a tutti i videogame con due o più 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**).

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** di diversi processi per funzioni diverse di una stessa applicazione, come per esempio il correttore ortografico di Word (posso continuare a scrivere mentre correggo, cioè posso compiere più azioni contemporaneamente);
- ▷ **condivisione** delle informazioni.

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**).

### ESEMPIO

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**).

Possiamo ora dare una definizione per le due situazioni di **interazione tra processi**:



### PROCESSI COOPERANTI

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



### PROCESSI DI COMPETIZIONE

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

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

- ▷ modello di computazione **indipendente**;
- ▷ modello di computazione con **cooperazione**;
- ▷ modello di computazione con **competizione**

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:

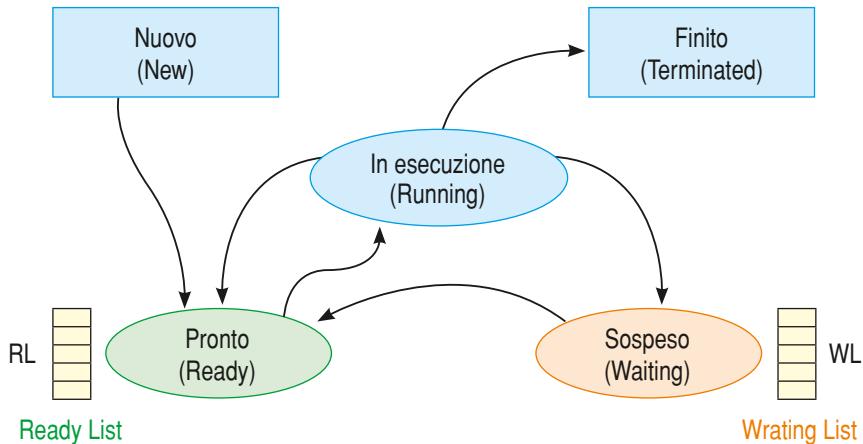
- ▷ **nuovo (new)**: è lo stato in cui si trova un processo appena è stato creato, cioè l'utente richiede l'esecuzione di un programma che risiede sul disco;
- ▷ **esecuzione (running)**: il **processo** sta evolvendo, nel senso che la **CPU** sta eseguendo le sue istruzioni, e quindi a esso è assegnato il processore. Nei sistemi a monoprocesso (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 assegna);
- ▷ **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) quando 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 routine di risposta all'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**.

## ■ 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 memoria del nuovo processo il segmento dati di sistema in modo da avere due copie di segmenti identiche, dato che il “figlio” è un clone del “padre”: l'unica differenza è il valore restituito dalla **fork()** stessa.



◀ \*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**. ▶

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.

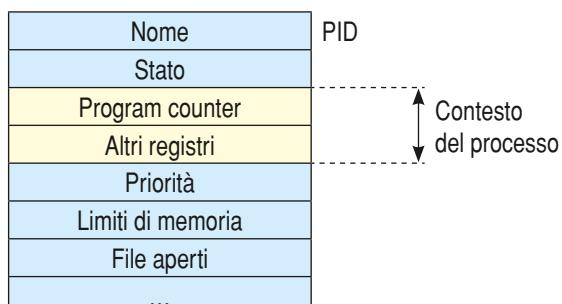
Il **SO** rilascia il **PID** del figlio e rimuove il suo descrittore di processo.

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.

## ■ 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.



## Prova adesso!

- Processi attivi in esecuzione
- Priorità dei processi

Individua sul tuo computer l'elenco dei processi attivi e individua le regole di attribuzione della priorità ai processi stessi.

Li puoi individuare a partire dal **Pannello di controllo** tra gli *Strumenti di amministrazione* (System information – Attività in esecuzione).

# 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 “accaparrarsene” 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** necessita di una **risorsa** generalmente non può richiedere *una particolare risorsa* ma solo una “generica” istanza di quella 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

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 e **competono** per poterlo avere a disposizione. 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.

Ma la **condivisione** offre anche un meccanismo che permette ai **processi** di scambiarsi delle informazioni: se per esempio due **processi** condividono una **area di memoria** dove sono definite alcune variabili, possiamo “trasferire” tramite esse il risultato delle elaborazioni del primo processo al secondo processo, cioè possiamo far **cooperare i processi**, e questo tipo di meccanismo prende il nome di **modello di cooperazione a memoria comune**.

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 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 necessita immediatamente di quella **risorsa** e se non gli viene assegnata immediatamente (in quanto occupata e quindi già in situazione di utilizzo da parte di altri 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 devono 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 **prerilasciabile** 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 preemptive** o **rilasciabile** come



### RISORSA PREEMPTIVE O PRERILASCIABILE

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.

I discorsi che abbiamo fatto si riferiscono alle **risorse** che i **processi** devono **condividere**, cioè **risorse comuni**: un **processo** può inoltre avere delle **risorse private** che esulano da questa trattazione dato che vengono assegnate da **SO** in modo esclusivo al **processo** stesso e non sono neppure visibili agli altri processi e quindi non richiedono di essere gestite in condivisione.

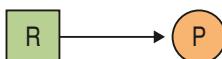
## ■ Grafo di Holt

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 risorse, 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.



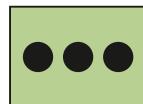
In questo modo si realizza un **grafo orientato diretto** (◀ **directed graph** ▶), con gli archi che hanno una sola direzione, e **bipartito**, in modo che non esistano archi che collegano nodi dello stesso sottoinsieme: gli archi possono solo connettere **nodi di tipo diverso**.

◀ **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. ▶



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**).

Un esempio di risorsa con molteplicità 3 è riportato sotto:

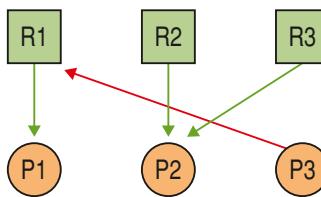


### ESEMPIO

Nel primo esempio abbiamo tre processi (p1, p2 e p3) e tre risorse (R1, R2 e 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
P2 richiede R3          //gli viene assegnata
P3 richiede R1          //NON gli viene assegnata, P3 rimane in attesa
```

La rappresentazione mediante il **grafo di Holt** è la seguente:



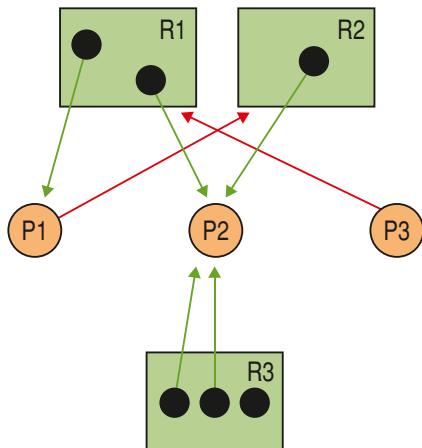
Supponiamo ora di avere classi di risorse con molteplicità diversa, per esempio:

- ▶ 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
```

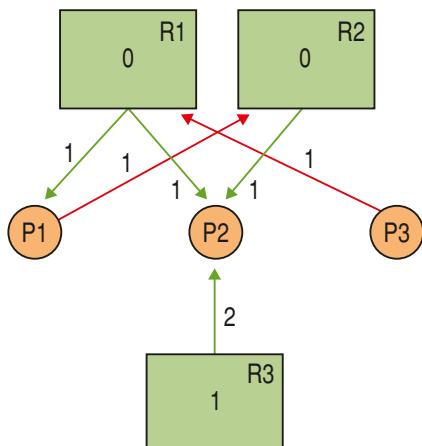
La rappresentazione mediante il **grafo di Holt** è quindi:



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.

### ESEMPIO

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**).

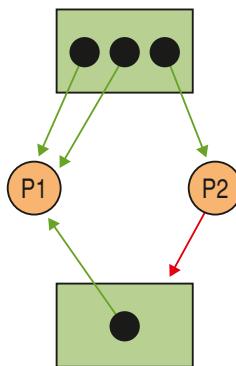


### GRAFO RIDUCIBILE

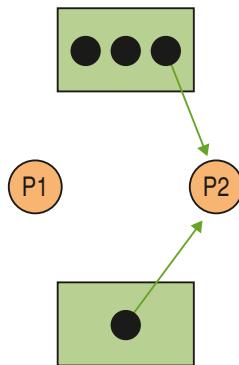
Un grafo di **Holt** si dice **riducibile** se esiste almeno un nodo di tipo processo con solo archi entranti.

### ESEMPIO

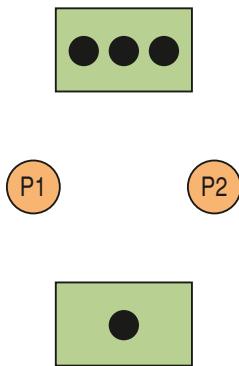
Nel seguente esempio abbiamo il processo P1 che sta utilizzando tre risorse e, dato che non è in attesa di altre, sta sicuramente evolvendo e, sicuramente, presto rilascerà quanto sta utilizzando:



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:



# 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 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**.

Inoltre questo codice viene condiviso con gli altri **thread** ed eseguito in parallelo agli altri **thread** mandati in esecuzione dallo stesso **processo** con il vantaggio di **condividere lo spazio di indirizzamento** e quindi le **strutture dati** e le **variabili**.

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**.

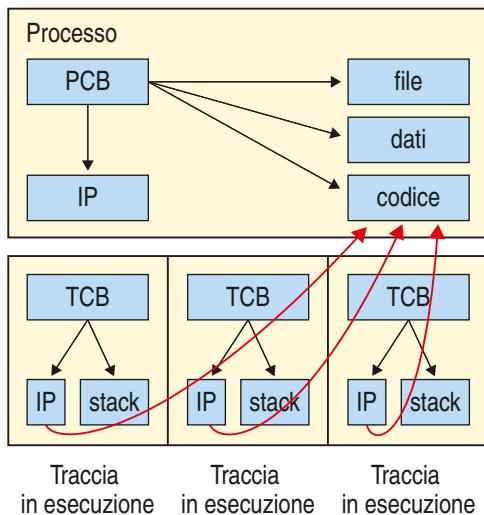
Per evolvere **parallelamente** agli altri **thread** o **processi** che si contendono la **CPU**, il **thread** ha inoltre un insieme di propri elementi che lo caratterizzano, in analogia ai processi tradizionali, chiamato **TCB** (il **Thread Control Block**), costituito da:

- ▷ 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**;
- ▷ un puntatore al **PCB** del processo contenitore.

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.

Il **TCB** è quindi simile al **PCB** e contiene i **registri**, lo **stack**, le **variabili "locali"** e lo **stato esecuzione**: dati "globali" e **TCB** "locale" 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:



È anche possibile definire variabili "personalizzate" per ogni singolo **thread**, con accesso tramite **chiave**, così che nessun altro **thread** "fratello" o **processo** le possa né vedere né tanto meno modificare.

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 i **thread** condividono molti dati e quindi sono in numero minore quelli da salvare e ripristinare.



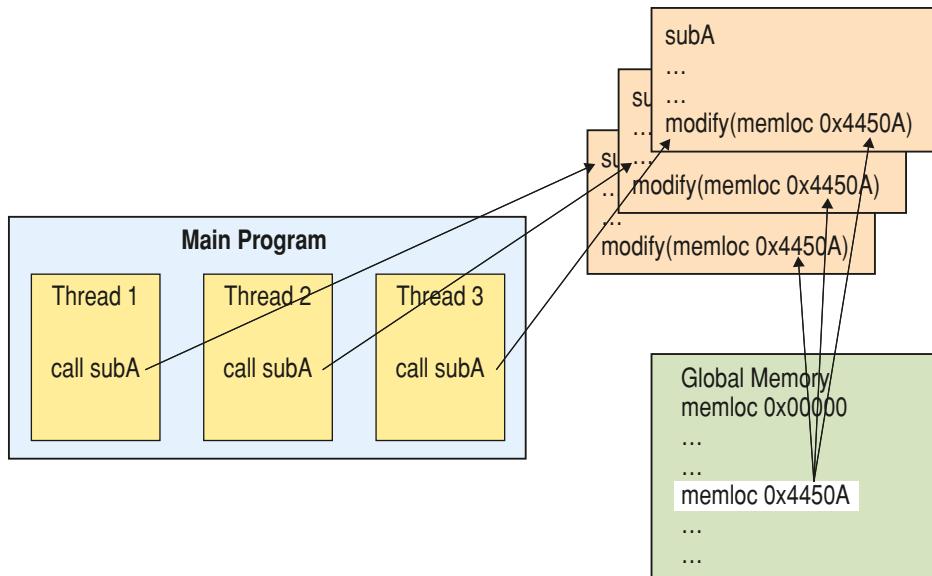
### THREAD SAFETY

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 **thread** garantendo che nessun **thread** possa accedere a dati in uno stato inconsistente, cioè durante il loro aggiornamento.

L'esecuzione dei **thread** richiede necessariamente che le routine di libreria debbano essere **rientranti**: i **thread** di un programma interagiscono con 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.

**ESEMPIO**

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

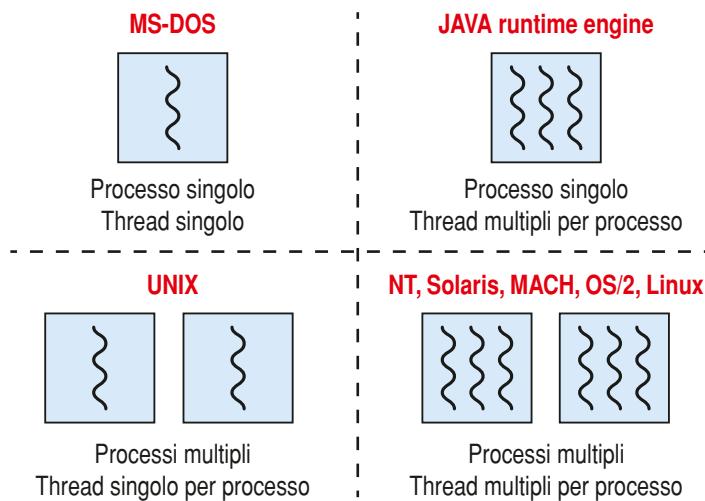
## ■ Soluzioni adottate: single threading vs multithreading

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**.



In questo volume faremo quindi riferimento sempre a situazioni con **thread multipli**, pensando a soluzioni con linguaggio **Java** a prescindere dal sistema operativo, e a soluzioni realizzate in **linguaggio C** eseguite su macchine con sistema operativo **Linux** (o in emulazione con **cygwin**).

## ■ 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 mediante 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 portabilità 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 l’ambiente **UNIX** che naturalmente non implementa i **thread**, ma questi vengono generati all’interno dei suoi processi.

## 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** 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.

## ■ 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**>: in essa viene definito il tipo **pthread\_t** che sarà usato per definire i **thread** nei programmi concorrenti creandoli all'interno di un processo, quindi siamo nella situazione 4 di **thread multipli per processo**.

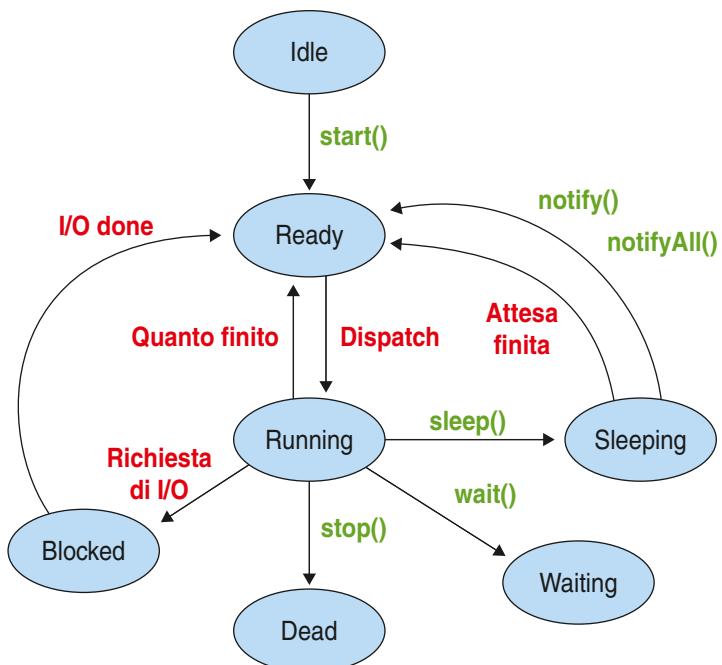
In alcuni compilatori la libreria base offerta dal **linguaggio C** include già le funzionalità relative alla gestione delle **espressioni regolari** che sono definite dallo standard **POSIX**: pertanto in queste situazioni non esiste propriamente una **libreria C** e una **POSIX**: a seconda del compilatore utilizzato è necessario verificare quando è necessario specificare in fase di compilazione l'inclusione della libreria precompilata per la gestione di quella certa funzionalità **POSIX**.

## ■ 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 a fianco.

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 reimpostazione 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**).

# 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 **Bohm 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 soprattut-

to 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.

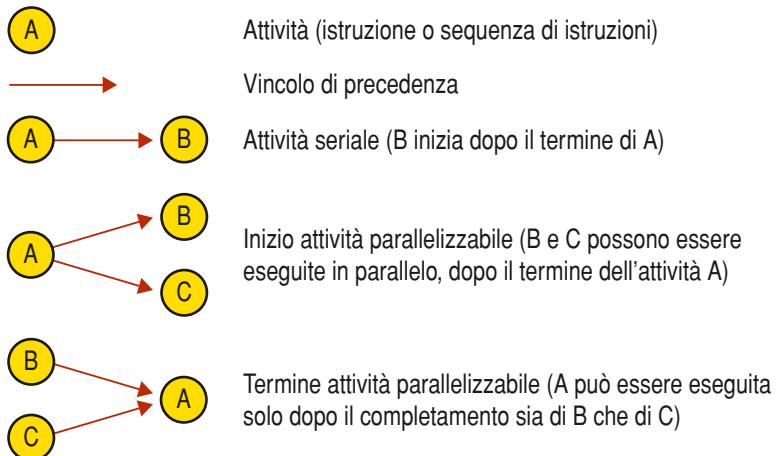
## ■ Processi non sequenziali e grafo di precedenza

Nei **processi sequenziali** la sequenza degli eventi che costituisce il processo è *totalmente ordinata*: se la rappresentiamo mediante un **grafo** orientato questo 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 nel 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 **Grafo delle precedenze**

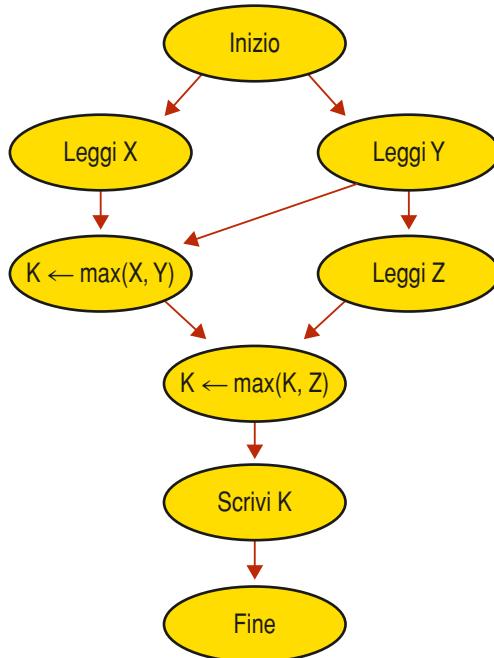
Vediamo un semplice esempio di un algoritmo che legge tre numeri e ne individua il maggiore. Per prima cosa scriviamo il codice sequenziale dell'algoritmo in *pseudocodifica (algoritmo sequenziale)*:

```

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.

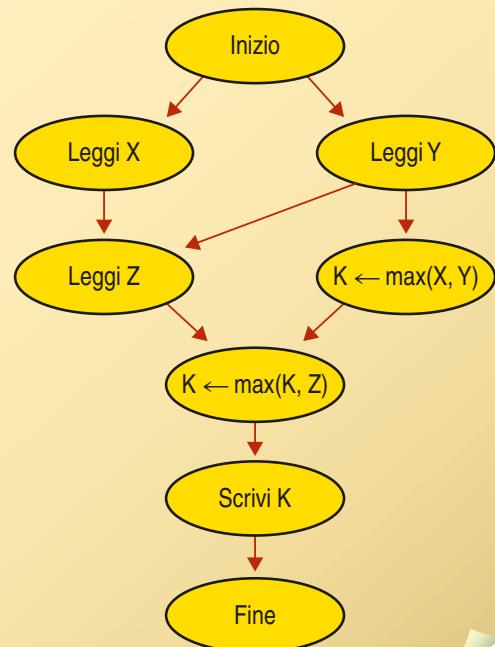
Il grafo è così fatto: ►



Questo non è l'unico diagramma delle precedenze possibili: 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**.



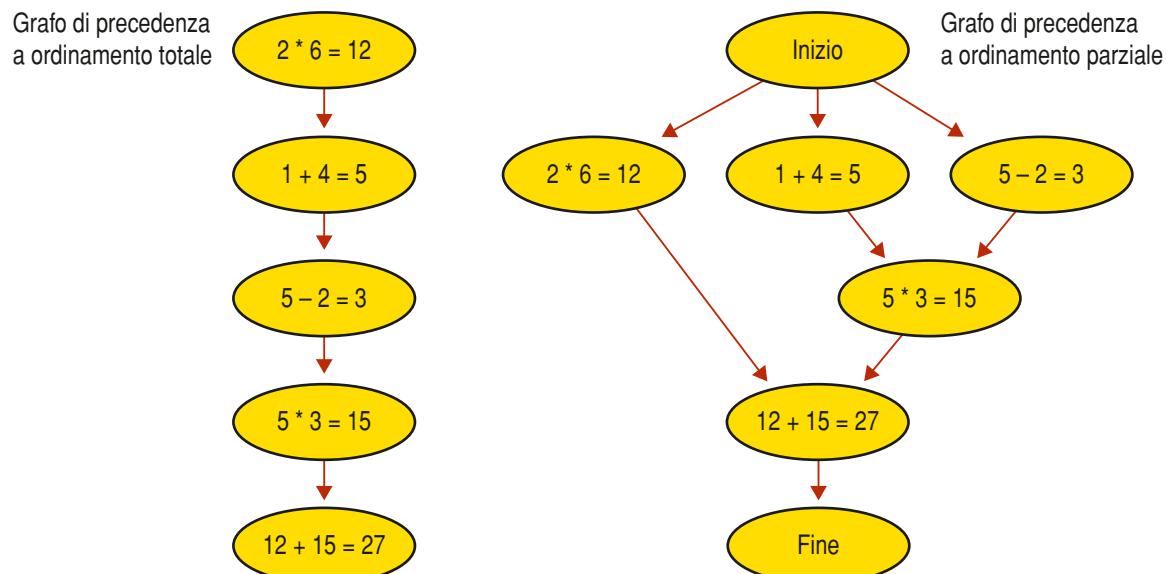
**ESEMPIO** ***Ordinamento parziale e totale***

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 di **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):


***Prova adesso!***

- Grafo delle precedenze
- Ordinamento parziale e totale

Date le seguenti espressioni algebriche:

$$1 \ 3+2-8*4-6/2+3 =$$

$$2 \ (3+2)-8*(4-6)/2+3 =$$

$$3 \ (3+2)-8*(4-6)/2+(3-8)*2 =$$

$$4 \ 3*2-(8*4-6)/2+(3-8/2)*2 =$$

**A** disegnare il grafo sequenziale a ordinamento totale;

**B** disegnare il grafo sequenziale a ordinamento parziale.

## ■ Scomposizione di un processo non sequenziale

Un **processo non sequenziale (parallelo)** 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      *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

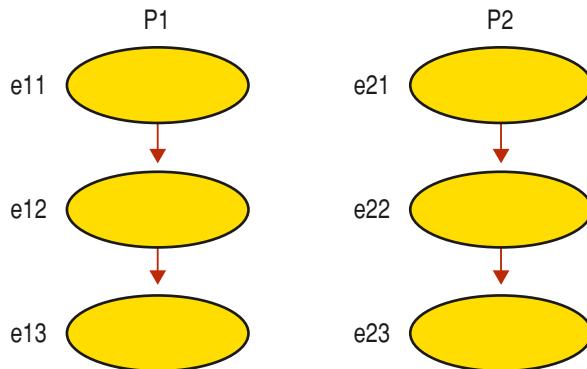
1. leggi X;
3. K ← SQRT(X);
5. scrivi K;

#### segmento 2

2. leggi Y;
4. W ← log(Y);
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.

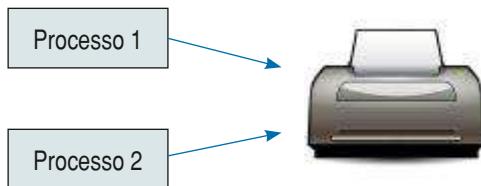
### A processi **totalmente ignari**

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 *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** esplicativi.

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

## Meccanismi di comunicazione e sincronizzazione tra entità (anticipazioni)

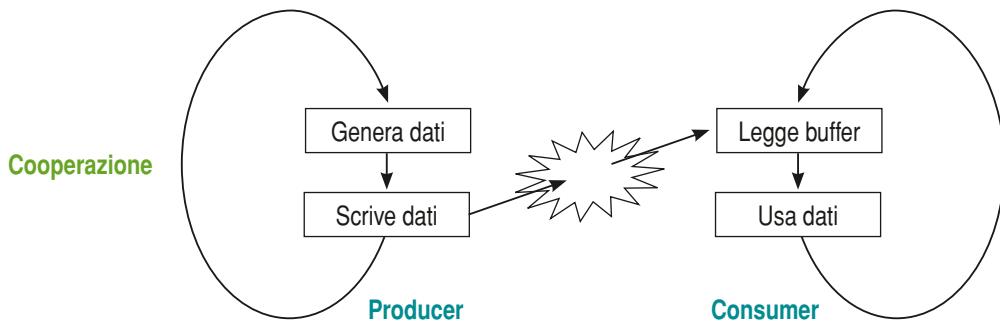
Negli ultimi due casi i **processi** devono **cooperare**, quindi il sistema operativo deve offrire i **meccanismi di sincronizzazione** in modo da garantire il corretto funzionamento regolando e gestendo i vincoli sull'ordine con cui devono essere eseguite le operazioni.

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)
```

# 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:

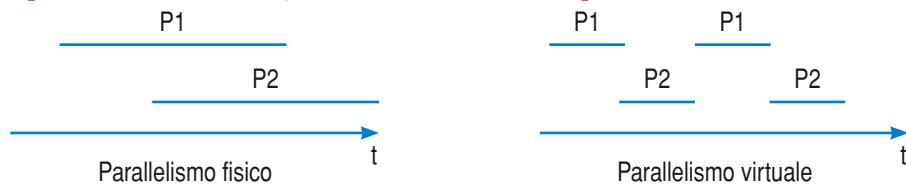
- ▶ un **elaboratore non sequenziale**;
- ▶ un **linguaggio di programmazione non sequenziale**.

### 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

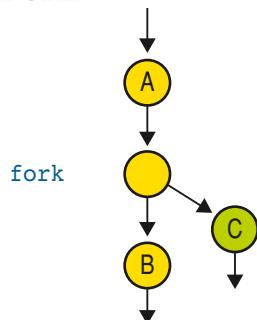
Un processo non sequenziale può essere scomposto in un insieme di processi sequenziali che possono essere eseguiti contemporaneamente.

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 riferimento 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.
```

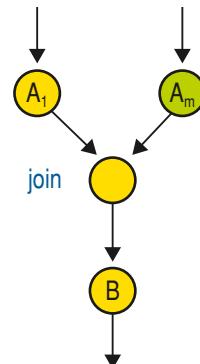
```
/* codice nuovo processo:*/
void figlio1()
{
    C: <istruzioni>;
}
```

## 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:

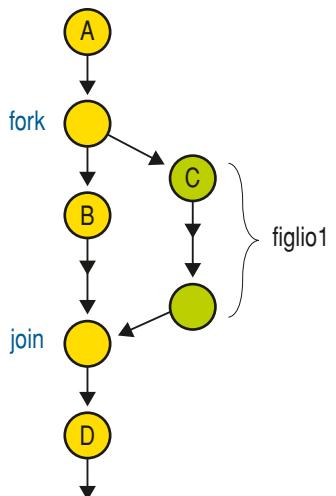
```
...
join p2;
B:<istruzioni>
...
```



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

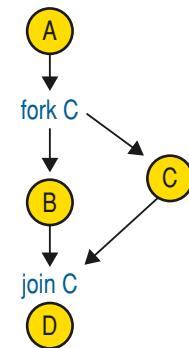
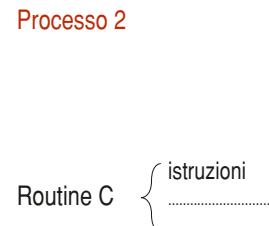
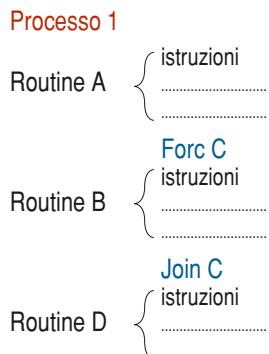
```
/* processo padre: */
inizio
...
    A:<istruzioni>;
    p2 = fork figlio1; // inizia l'elaborazione parallela
    B: <istruzioni>;
    join p2;           // termina l'elaborazione parallela
    D:<istruzioni>;
...
fine

/* 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.



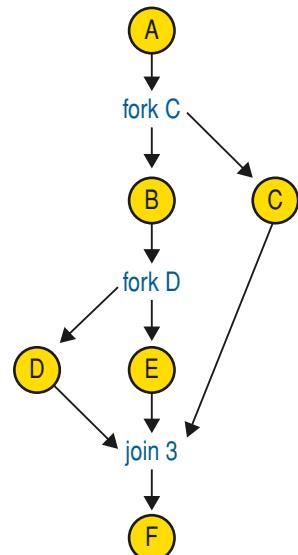
## Join (count)

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.

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.

```

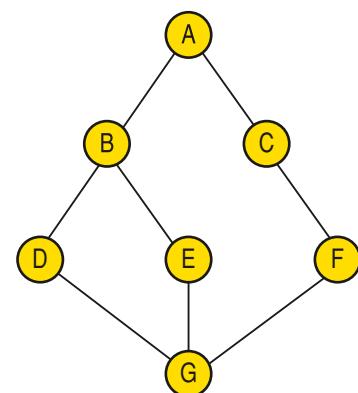
Nella letteratura la sintassi della pseudocodifica a volte è leggermente differente da quella appena presentata: spesso viene semplificata omettendo le <istruzioni> e indicando semplicemente con le lettere maiuscole A,B,C oppure con S1,S2,..., Sn il blocco o la sequenza di istruzioni, e tale simbologia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

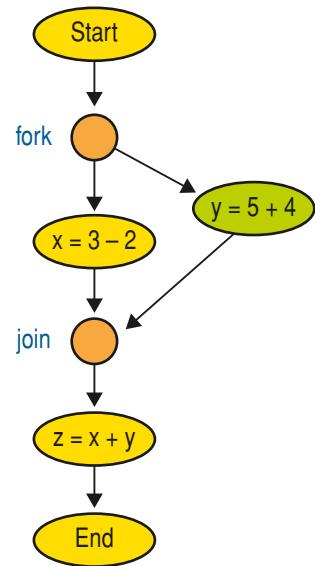
Sono però concettualmente diverse dato che sono *chiamate di sistema* mentre quelle da noi descritte sono istruzioni di un ipotetico *linguaggio concorrente di alto livello*: possono essere comunque utilizzate didatticamente per realizzare programmi concorrenti e alcune codifiche in **linguaggio C** 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

Scriviamo un programma parallelo che esegue la seguente espressione matematica:

$$z = (3-2) * (5+4)$$

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:

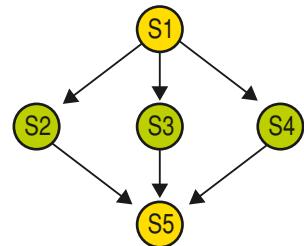
```

cobegin
  <elenco delle attività parallele>
coend
  
```

**ESEMPIO**

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
```

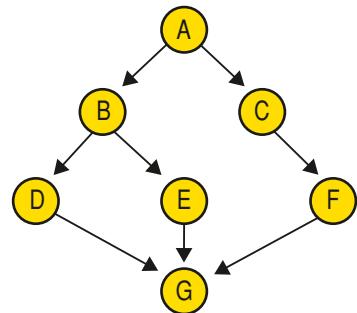


Al termine dell'esecuzione della sequenza S1 vengono attivati tre processi che eseguono in parallelo le sequenze di istruzioni S2, S3, S4: il processo padre S1 si sospende e rimane in attesa della loro terminazione.

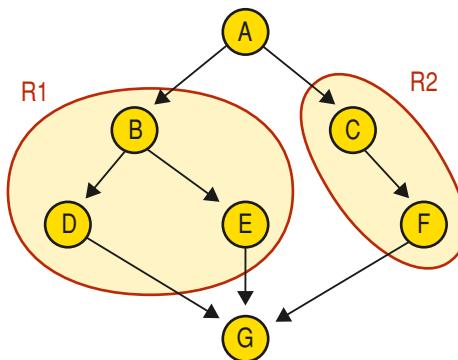
All'interno dei singoli processi possono a loro volta essere presenti delle istruzioni di **cobegin-coend**, cioè è possibile avere l'annidamento di più costrutti **cobegin-coend**.

**ESEMPIO**

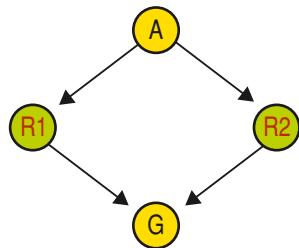
Codifichiamo il grafo delle precedenze utilizzando il costrutto **cobegin-coend** della figura a fianco. ►



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
<b>inizio</b> A <b>cobegin</b> R1 R2 <b>coend</b> G <b>fine</b>	<b>inizio</b> B <b>cobegin</b> D E <b>coend</b> <b>fine</b>	<b>inizio</b> C F <b>fine</b>

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.

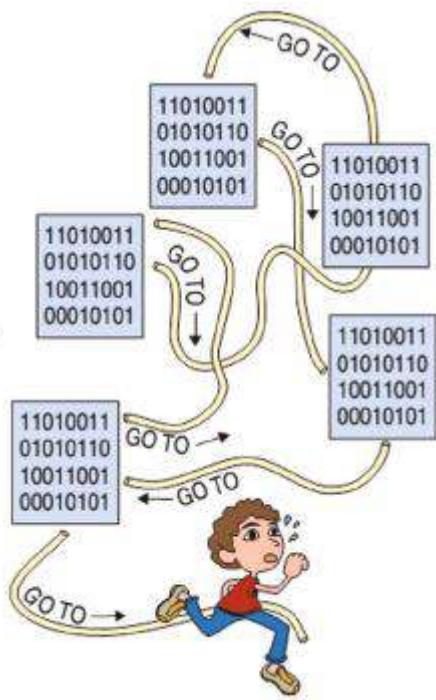
◀ **Spaghetti code** Spaghetti code is a derogatory term for computer programming that is unnecessarily convoluted, and particularly programming code that uses frequent branching from one section of code to another (using many GOTOs or other “unstructured” constructs). Spaghetti code sometimes exists 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**.



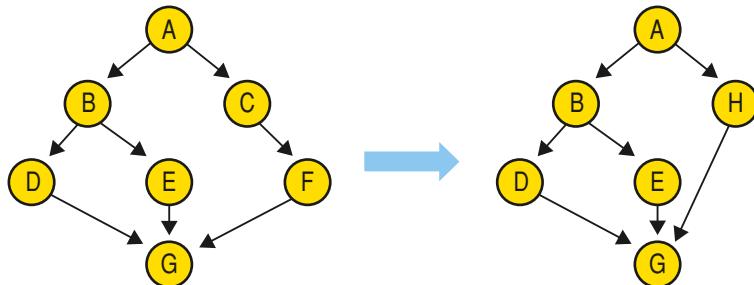
### EQUIVALENZA TRA FORMALISMI

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

#### ESEMPIO

#### **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:

```

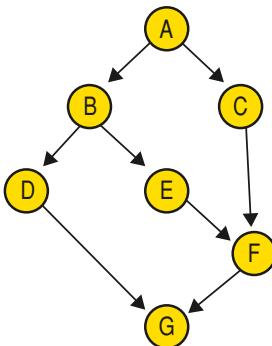
/* processo padre: */
inizio
...
A
p2 = fork(H);           // inizia l'elaborazione parallela di B e H
    B
    p4 = fork(E);       // inizia l'elaborazione parallela di D e E
        D
        join p4;         // termina l'elaborazione parallela di D e E
    join p2;             // termina l'elaborazione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

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

```

**ESEMPIO*****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 'sottografo' 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:

```

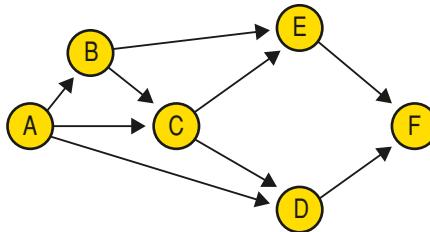
/* processo padre: */
inizio
...
A
p2=fork(C);           // inizia l'elaborazione parallela di B e C
  B
  P3=fork(E);         // inizia l'elaborazione parallela di D e E
    D
    goto L1
  join C;            // aspetta la terminazione di C
L1:join E;            // aspetta la terminazione di E join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

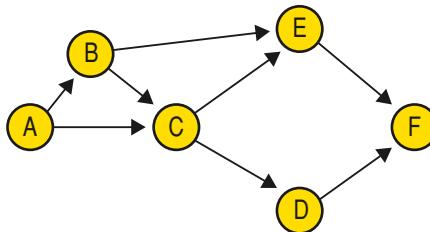
## ■ Semplificazione delle precedenze

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**.

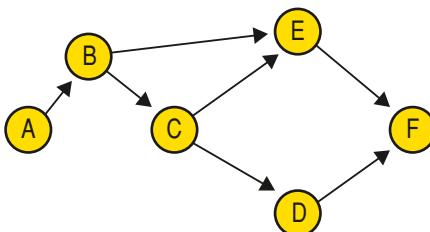
In presenza di **precedenze implicite** è possibile togliere un arco in quanto questo risulta ridondante ed è quindi possibile semplificare il grafo: vediamo un esempio e semplifichiamo il grafo riportato nella figura seguente:



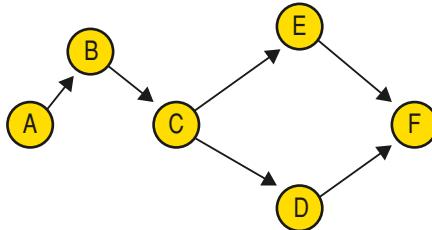
Possiamo osservare che il nodo D ha come precedenza il nodo A e anche il nodo C ha come precedenza il nodo A: quindi il nodo D ha il nodo A come precedenza diretta ma deve attendere l'elaborazione di C che a sua volta dipende da A; è possibile eliminare la precedenza tra A → D che risulta essere implicita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



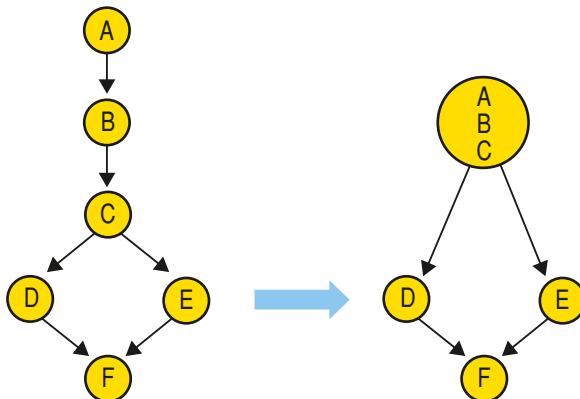
Analogo discorso può essere fatto tra B, che ha come precedenza il nodo A, e il nodo C, che anch'esso ha come precedenza il nodo A: quindi il nodo B ha il nodo A come precedenza diretta e dato che deve attendere l'elaborazione di B che a sua volta dipende da A è possibile eliminare la precedenza tra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

# ESERCITAZIONI DI LABORATORIO 3

## LA FORK IN C



I codici sorgenti sono nel file [C\\_fork.rar](#) scaricabile dalla cartella materiali nella sezione del sito [www.hoepliscuola.it](http://www.hoepliscuola.it) riservata al presente volume.

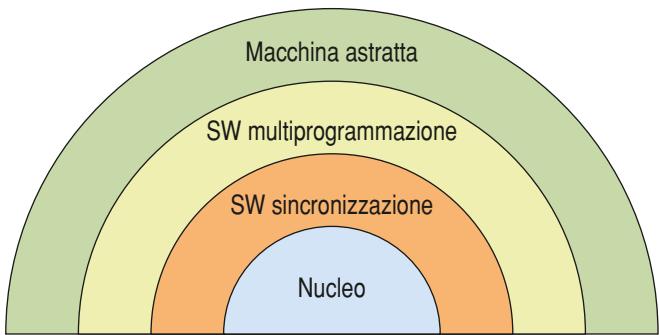
### ■ Macchina astratta

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 monoprocessoressi 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 esercitazione inizieremo a realizzare gerarchie di processi mediante l'istruzione **fork** e nella prossima esercitazione implementeremo i costrutti fondamentali descritti ad alto livello nella lezione 5 della corrente unità di apprendimento:

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

## ■ L'istruzione fork

Scriviamo ora il primo programma che genera un processo figlio eseguendo una operazione di **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 vengono duplicati nel figlio al momento della esecuzione **fork**.

Il processo figlio viene creato con questa istruzione e viene mandato in esecuzione proprio dall'istruzione successiva: quindi dopo l'istruzione e **fork** avremo due processi in evoluzione, il padre e il figlio.

La **fork** ha come parametro di ritorno un dato **integer** che ha valori differenti nei due processi:

- nel processo padre è il **PID** del processo appena creato;
- nel processo figlio è uguale a 0.

Solo la variabile **pid** di ritorno dalla **fork** ha valori diversi nel padre e nel figlio in quanto il processo figlio è una "copia perfetta" del processo padre, inclusa anche la sua area dati: quindi anche il processo figlio procede l'elaborazione con gli esiti delle elaborazioni del processo padre.

È possibile sfruttare il valore della variabile **pid** per distinguere il padre dal figlio in modo da far eseguire a ciascuno le istruzioni desiderate: dato che entrambi condividono il codice, 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)
```

questa 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 inseriamo le istruzioni specifiche che devono essere eseguite o dal padre oppure dal figlio.

## Terminazione di un processo

Un processo può terminare:

- **involontariamente**, se per esempio cerca di effettuare operazioni illegali oppure mediante una interruzione;

- **volontariamente**, o perché ha finito tutte le istruzioni oppure con la chiamata alla funzione **exit()**.

La funzione **exit()** ha la seguente sintassi:

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

Prevede un parametro (**status**) mediante il quale il **processo** che termina può comunicare al padre informazioni sul suo stato di terminazione; generalmente è un intero a 16 bit ed ha i seguenti significati:

- se il byte meno significativo di **status** è zero, il più significativo rappresenta lo stato di terminazione (**terminazione volontaria**, per esempio il valore indicato nella funzione **exit**);
- in caso contrario, il byte meno significativo di **status** descrive il segnale che ha terminato il figlio (**terminazione involontaria**) e il suo valore viene “forzato” dal sistema operativo.

Se il valore di **status** è minore di 256 la terminazione è volontaria e, quindi, l'elaborazione è corretta e il valore di **status** è quello che viene passato dal figlio come parametro: è possibile utilizzare il secondo byte per comunicare dati al processo padre sempre tenendo presente che il range dei valori possibili si limita a 0-255.

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

Codice eseguito dal processo padre	Codice eseguito dal processo figlio
<pre>int main(){     int pid;     printf( "1) prima della fork \n" );     pid = fork(); // creo processo figlio ---&gt;     printf( " 2) dopo della fork \n" );     if (pid == 0){         printf( " 3) sono il processo figlio\n" );         exit(1) ;     }     else{         printf( " 3) sono il processo padre\n" );         exit(0) ; // termina padre     } }</pre>	<pre>int main(){     int pid;     printf( "1) prima della fork \n" );     pid = fork();     printf( " 2) dopo della fork \n" );     if (pid == 0){         printf( " 3) sono il processo figlio\n" );         exit(1) ;     }     else{         printf( " 3) sono il processo padre\n" );         exit(0) ; // termina padre     } }</pre>

Completiamo la codifica inserendo le direttive per il compilatore:

```
fork1.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int pid;
6     printf( "1) prima della fork \n" );
7     pid = fork(); // creo processo figlio
8     printf( " 2) dopo della fork \n" );
9     if (pid == 0){
10         printf( " 3) sono il processo figlio\n" );
11         exit(1) ; // termina il processo figlio
12     }
13     else{
14         printf( " 3) sono il processo padre\n" );
15         exit(0) ; // non necessaria
16     }
17 }
```

Avviandone l'esecuzione otteniamo il seguente risultato sullo schermo:

```

Paolo@PCwin8 ~/ua1
$ gcc fork0.c -o fork0
Paolo@PCwin8 ~/ua1
$ ./fork0
1) prima della fork
2) dopo della fork
3) sono il processo padre
2) dopo della fork
3) sono il processo figlio
Paolo@PCwin8 ~/ua1
$ 
```

Si vede chiaramente come l'istruzione 8 venga eseguita due volte, sia dal processo padre che dal processo figlio: successivamente i due processi “scelgono” il ramo opportuno dell'istruzione **if** di riga 9 in base al valore della variabile **pid** che, ripetiamo, per il processo figlio ha valore 0.



## Prova adesso!

- 1 Inserisci alcune variabili nel programma e assegna loro valori a piacere prima e dopo l'istruzione **fork**.
- 2 Modifica ora il programma facendo il test non sul **pid == 0** ma confrontandolo con il valore che viene generato alla sua creazione, in modo da “invertire” i due rami della selezione.
- 3 All'interno dei due rami, oltre alla frase di saluto visualizza anche il valore del **pid** utilizzando la funzione **getpid()**, in modo da ottenere un output simile al seguente:

```

Paolo@PCwin8 ~/ua1
$ ./fork0Sol
1) prima della fork
2) dopo della fork
2) dopo della fork
3) sono il processo padre pid: 6392
4) la mia variabile uguale contiene : 10
5) sono il processo diversa contiene: 30
3) sono il processo figlio con pid: 5324
4) la mia variabile uguale contiene : 10
5) sono il processo diversa contiene: 20
Paolo@PCwin8 ~/ua1
$ 
```

Confronta il tuo codice con quello riportato nel file **fork0Sol.c**.

## ■ PID del padre e del figlio

Ogni processo figlio “si ricorda” il **PID** del **processo** padre che prende il nome di **parent pid** o **PPID**. Il linguaggio C mette a disposizione due funzioni che permettono di sapere per il processo corrente quale è il proprio **PID** e quale è il **PID** del suo genitore:

```
int getpid()           //PID proprio
int getppid()          //PID del padre
```

Utilizziamole per completare l'esempio precedente:

```
fork2.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int pid;
6     printf( "1) prima della fork \n" );
7     pid = fork();           // creo processo figlio
8     printf( "2) dopo della fork \n" );
9     if (pid == 0){
10         printf( " 3) sono il processo figlio con pid:%d.", getpid() );
11         printf( " Il mio papi ha pid: %d\n", getppid() );
12         exit(1);           // termina il processo figlio
13     }
14     else{
15         printf( " 3) sono il processo padre con pid:%d.", getpid() );
16         printf( " Il mio papi ha pid: %d\n", getppid() );
17         exit(0);           // non necessaria
18     }
19 }
```

Avviandone l'esecuzione otteniamo il seguente risultato sullo schermo:

```
Paolo@PCwin8 ~/u1/c_fork
$ gcc fork2.c -o fork2
$ ./fork2
1) prima della fork
2) dopo della fork
3) sono il processo figlio con pid:9604. Il mio papi ha pid: 5484
3) sono il processo padre con pid:5484. Il mio papi ha pid: 4620
```

È anche possibile visualizzare **PID** e **PPID** su tutti i processi che sono in esecuzione digitando da console il comando:

```
ps -e
```

PID	PPID	PGID	WINPID	TTY	UID	STIME	COMMAND
4092	10336	4092	9256	pty0	197609	15:58:44	/usr/bin/ps
10336	4368	10336	2220	pty0	197609	15:57:53	/usr/bin/bash
4368	1	4368	4368	?	197609	15:57:53	/usr/bin/mintty

Modifichiamo ora il codice aggiungendo una istruzione del figlio rispetto al padre; a tal fine utilizziamo la seguente funzione

```
void sleep(int secondi)
```

che “addormenta” il processo per un numero parametrico di secondi. Inseriamo quindi l’istruzione 11:

```

8     printf( " 2) dopo della fork su %d , getpid() );
9     if (pid == 0){
10       printf( " 3) sono il processo figlio con pid:%d.", getpid() );
11       sleep(3);           // ritardo di 3 secondi
12       printf( " Il mio papi ha pid: %d\n", getppid());
13       exit(1);           // termina il processo figlio
14   }

```

Aggiungiamo a ogni istruzione la funzione `getpid()` in modo da poter individuare il processo che la esegue e mandiamo in esecuzione il programma ottenendo:

```

Paolo@PCwin8 ~/ua1/l3_c_fork
$ gcc Fork3.c -o fork3
Paolo@PCwin8 ~/ua1/l3_c_fork
$ ./fork3
1) prima della fork 5992
2) dopo della fork 5992
3) sono il processo padre con pid:5992. Il mio papi ha pid: 5996
2) dopo della fork 3208

Paolo@PCwin8 ~/ua1/l3_c_fork
$ 3) sono il processo figlio con pid:3208. Il mio papi ha pid: 1

```

Osserviamo come il **processo padre** termina prima del **processo figlio** lasciandolo, di fatto, **orfano**: lo possiamo riconoscere dal fatto che il programma principale è terminato e sullo schermo appare il **prompt** dei comandi e solo in un successivo tempo viene visualizzata la riga scritta dal processo figlio con l’istruzione 12.

Inoltre il processo figlio in questa riga “dichiara” di avere come padre il processo 1, ma questo è il **PID** del processo **init ()** che lo ha “adottato”.

Vedremo come rimediare a questa situazione nelle prossime lezioni.



### Prova adesso!

- Istruzione `fork`
- Identificatore di processo `PID`

Scrivi un programma dove un padre genera un numero di processi figli definiti dall’operatore mediante la lettura di un valore intero mediante una funzione `faiFiglio()`.

Visualizza per ciascuno di essi il proprio **PID** e quello del padre.

Confronta la tua soluzione con quella presente nel file **fratelliSol.c**.

Quindi prova a togliere l’istruzione `exit(0)` all’interno del segmento eseguito dal figlio: cosa ti aspetti che venga visualizzato? Perche?

# ESERCITAZIONI DI LABORATORIO 4

## FORK ANNIDATE ED ESECUZIONE NON DETERMINISTICA



### Info

I codici sorgenti sono nel file **C\_fork.rar** scaricabile dalla cartella materiali nella sezione del sito [www.hoepliscuola.it](http://www.hoepliscuola.it) riservata al presente volume.

Riprendiamo il codice dell'esercizio **fork2.c** e modifichiamolo in modo da effettuare tre **fork()**, come si può osservare nel seguente programma alle istruzioni 5, 6 e 7:

```
fork4.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int pid1, pid2, pid3;
5     pid1 = fork();                                // creo processo figlio
6     pid2 = fork();                                // creo processo figlio
7     pid3 = fork();                                // creo processo figlio
8     if ((pid1 == 0) || (pid2 == 0) || (pid3 == 0)) { // se uno è 0 è un figlio
9         printf(" Sono il processo figlio con pid:%d.", getpid());
10        printf(" Il mio papi ha pid: %d\n", getppid());
11        sleep(1);                                 // attesa per non creare orfani
12        exit(1);                                 // termina il processo figlio
13    }
14    else{
15        sleep(2);
16        printf(" Sono il processo padre con pid:%d. \n", getpid());
17    }
18    return 0;
19 }
```



### 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?

Una esecuzione produce il seguente output, dove sono stati colorati i PID uguali per meglio individuarli:

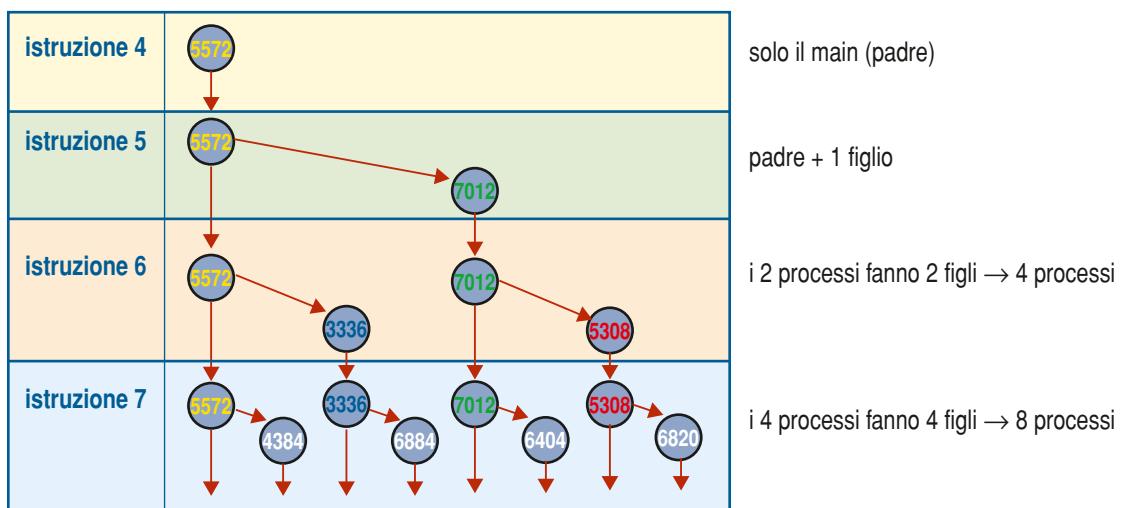
```

Paolo@PCwin8 ~/ua1/13_c_fork
$ ./fork4
Sono il processo figlio con pid:4384. Il mio papi ha pid: 5572
Sono il processo figlio con pid:5308. Il mio papi ha pid: 7012
Sono il processo figlio con pid:6820. Il mio papi ha pid: 5308
Sono il processo figlio con pid:3336. Il mio papi ha pid: 5572
Sono il processo figlio con pid:6884. Il mio papi ha pid: 3336
Sono il processo figlio con pid:7012. Il mio papi ha pid: 5572
Sono il processo figlio con pid:6404. Il mio papi ha pid: 7012
Sono il processo padre con pid:5572.

Paolo@PCwin8 ~/ua1/13_c_fork
$ 
```

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 5) genera un processo che a sua volta esegue la seconda **fork()** (istruzione 6), quindi i primi due processi generano altri due figli che a loro volta eseguiranno la **fork()** della istruzione 7: in totale sono (padre + figlio) + 2 processi + 4 processi per un totale di otto processi ( $2^3$  processi).

Se quindi si facessero 10 **fork()** di seguito si genererebbero  $2^{10}$  figli!

Trasformiamo ora il programma che genera più figli in modo da creare una gerarchia di N livelli, cioè dove viene letto come parametro il numero di antenati desiderati e ciascuno crea il proprio discendente passandogli il parametro decrementato.

A tal fine effettuiamo una **chiamata ricorsiva** all'interno del segmento di codice **faiFiglio()**, in modo da richiamare la funzione che esegue la **fork**:

```

10  void faiFiglio (int quanti){
11    int pid;
12    pid = fork();           // creo processo figlio
13    if (pid == 0){
14      mettiSpazi(quanti);
15      printf( "Sono il processo figlio con pid:%d.", getpid() );
16      printf( "Il mio papi ha pid: %d\n", getppid() );
17      if (quanti > 0)
18        faiFiglio(quanti-1); // il figlio diventa il padre
19      else
20        exit(0);
21    }
22    else{
23      mettiSpazi(quanti);
24      printf( "Sono il processo padre con pid:%d.\n", getpid() );
25    }
26  }

```

Per meglio visualizzare sullo schermo la gerarchia introduciamo una semplice funzione che indenta gli output che ogni processo produce in modo da “visualizzare anche graficamente” la gerarchia.

```

[*] gerarchia.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define NRFIGLI 4
4  void mettiSpazi(int quanti){ // per meglio vedere la gerarchia
5    int x;
6    for (x = 0; x <= quanti; x++)
7      printf(" ");
8  }

```

Avviandone l'esecuzione otteniamo il seguente risultato sullo schermo:

```

Paolo@PCwin8 ~/ua1
$ ./gerarchia
Sono il processo padre con pid:32.
Termino elaborazione processo con pid:32.
Sono il processo figlio con pid:8072.Il mio papi ha pid: 32

Paolo@PCwin8 ~/ua1
$ Sono il processo padre con pid:8072.
$ Termino elaborazione processo con pid:8072.
$ Sono il processo figlio con pid:5612.Il mio papi ha pid: 8072
$ Sono il processo padre con pid:5612.
$ Termino elaborazione processo con pid:5612.
$ Sono il processo figlio con pid:4888.Il mio papi ha pid: 5612
$ Sono il processo padre con pid:4888.
$ Termino elaborazione processo con pid:4888.
$ Sono il processo figlio con pid:6916.Il mio papi ha pid: 4888
$ Sono il processo padre con pid:6916.
$ Termino elaborazione processo con pid:6916.
$ Sono il processo figlio con pid:5236.Il mio papi ha pid: 6916

```

Osserviamo come l'output sullo schermo evidensi il fatto che il `main()` termina prima dei processi figli che lui stesso ha generato.



## Prova adesso!

- Istruzione fork ricorsiva
- Identificatore di processo PID

Per ottenere un output "ordinato" è necessario introdurre dei ritardi nella esecuzione dei processi, in modo che i padri "attendano" la terminazione dei figli.

Confronta la tua soluzione con quella presente nel file `gerarchiaSol.c`.

**Attenzione!** Con l'introduzione della funzione `sleep()` **NON** stiamo effettuando alcuna **sincronizzazione**: abbiamo semplicemente rallentando il processo padre in modo che termini dopo il proprio processo figlio.

I meccanismi di **sincronizzazione** verranno descritti a partire dalla prossima lezione.

## Esecuzione non deterministica

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



## Prova adesso!

- Generazione processi
- Esecuzione sequenziale

Scrivi un programma concorrente dove un processo padre genera solamente due processi figli numerando ogni istruzione di output in modo da poterne "seguire" la cronologia di esecuzione sullo schermo.

Mandalo in esecuzione più volte confrontando il risultato.

Una possibile implementazione è riportata di seguito:

```
fork5.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 void main(){
4     int pid, pid1, pid2;
5     pid1 = fork();
6     if(pid1 == 0){      // processo figlio
7         printf("1) sono il primo processo figlio con pid: %i\n", getpid());
8         exit(1);        // termina primo processo figlio
9 }
```

```

10 } else{
11     printf("2) sono il processo padre\n");
12     printf("3) ho creato un processo con pid: %i\n", pid1);
13     printf("4) il mio pid e' invece: %i\n", getpid());
14     pid2 = fork();
15 } if (pid2 == 0){ // secondo processo figlio
16     printf("5) sono il secondo processo figlio con pid: %i\n", getpid());
17     exit(2); // termina secondo processo figlio
18 }
19 } else {
20     printf("6) sono il processo padre\n");
21     printf("7) ho creato un secondo processo con pid: %i\n", pid2);
22 }
23 }
24 }
```

Una prima esecuzione dà il seguente output:

```

~/ua1/l3_c_fork
$ gcc fork5.c -o fork5
Paolo@PCwin8 ~/ua1/l3_c_fork
$ ./fork5
1) sono il primo processo figlio con pid: 1896
2) sono il processo padre
3) ho creato un processo con pid: 1896
4) il mio pid e' invece: 8152
5) sono il processo padre
7) ho creato un secondo processo con pid: 7904
5) sono il secondo processo figlio con pid: 7904
Paolo@PCwin8 ~/ua1/l3_c_fork
$ |
```

Una seconda esecuzione dà il seguente output:

```

~/ua1/l3_c_fork
Paolo@PCwin8 ~/ua1/l3_c_fork
$ ./fork5
2) sono il processo padre
3) ho creato un processo con pid: 1652
4) il mio pid e' invece: 100
1) sono il primo processo figlio con pid: 1652
6) sono il processo padre
7) ho creato un secondo processo con pid: 8160
5) sono il secondo processo figlio con pid: 8160
Paolo@PCwin8 ~/ua1/l3_c_fork
$ |
```

Possiamo osservare che entrambe le esecuzioni non rispettano l'ordinamento sequenziale delle istruzioni e ogni esecuzione ha un ordine diverso, confermando come la schedulazione giochi un ruolo fondamentale nella evoluzione dei processi.

Per ottenere esecuzioni deterministiche è necessario introdurre la sincronizzazione, come vedremo nella prossima lezione.



## Prova adesso!

- Generazione processi
- Esecuzione sequenziale

Scrivi un programma che utilizza l'istruzione di **switch** per intercettare l'esecuzione dei due processi prendendo il **PID** come selettore.

Confronta la tua soluzione con quella presente nel file **fork6.c**.

Successivamente modifica il programma **fork5.c** in modo che ciascun figlio generi un processo in modo da avere una gerarchia a tre livelli.

Manda in esecuzione più volte il programma osservando la sequenza non deterministica delle istruzioni.

Confronta la tua soluzione con quella presente nel file **fork5Sol.c**.



# ESERCITAZIONI DI LABORATORIO 5

## LE FUNZIONI WAIT() E WAITPID()



### Info

I codici sorgenti sono nel file **C\_fork\_join.rar** scaricabile dalla cartella materiali nella sezione del sito [www.hoepliscuola.it](http://www.hoepliscuola.it) riservata al presente volume.

### ■ La funzione wait()

Nella esercitazione precedente abbiamo visto come l'elaborazione parallela di più processi necessita di un meccanismo di sincronizzazione affinché il primo processo che termina il proprio compito possa "comunicare" agli altri il risultato della propria elaborazione altrimenti l'esecuzione avviene in modo NON deterministico, a causa delle politiche di schedulazione del **sistema operativo**.

Sia il costrutto **fork-join** che il **cobegin-coend** necessitano di un meccanismo mediante il quale un processo possa "aspettare" gli altri processi che non hanno ancora finito di produrre i loro risultati.

L'istruzione che permette a un processo padre di attendere e leggere lo stato di terminazione del processo figlio è la system call **wait()**, che ha la seguente sintassi:

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

Il parametro **status** è l'indirizzo della variabile in cui viene memorizzato lo stato di terminazione del figlio mentre il valore di ritorno è il **pid** del processo terminato oppure un codice di errore (<0): la variabile indirizzata da **status** è quindi utilizzata come secondo parametro di ritorno dato che viene letto dal processo padre.

`pid_t` è un tipo di dato definito nel linguaggio C per rappresentare l'ID del processo: sostanzialmente è un numero intero con segno.

Per utilizzare il tipo `pid_t` può essere necessario includere come **header files** la libreria `sys/types.h`

In generale il tipo `pid_t` viene utilizzato quando si vuole svincolare il codice dal tipo di macchina/sistema operativo oppure quando non si conosce come esso viene implementato, in quanto potrebbe essere un `short int`, un `int`, un `long int`, oppure un `long long int`: nella libreria **GNU** è un semplice `int`, e quindi la funzione può essere anche così scritta:

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

## AREA digitale



Why should pid\_t be used?

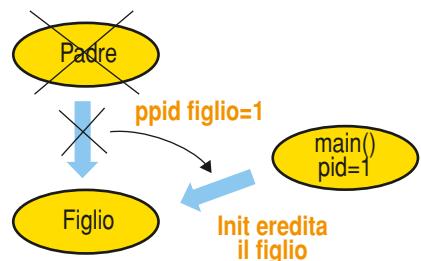
Vediamo gli effetti della chiamata di una `wait()`: dato che un processo che chiama la funzione `wait()` può avere figli in esecuzione abbiamo alcune possibili situazioni:

- se nessun figlio è terminato, il processo si sospende in attesa della *terminazione del primo di essi*;
- se almeno un figlio è già terminato, il processo padre NON si sospende per quel figlio e `wait()` ritorna immediatamente il suo stato di terminazione;
- se non esiste neanche un figlio, `wait()` NON è sospensiva e ritorna un codice di errore (valore ritornato < 0).

## Terminazioni particolari di un processo

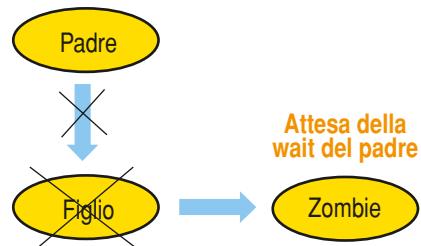
### A Terminazione del padre

Se il processo che termina ha figli in esecuzione, il processo `main()` adotta i figli dopo la terminazione del padre e nella process structure di ogni figlio al `pid` del processo padre viene assegnato il **valore 1**.



### B Terminazione del figlio

Se il processo termina prima che il padre ne rilevi lo stato di terminazione il figlio passa nello stato **zombie**, tranne quando si trova nella situazione precedente, cioè se è stato "adottato" dal `main()` che rileva automaticamente il suo stato di terminazione.



Cerchiamo di comprendere le diverse situazioni mediante esempi.

## AREA digitale



Rilevazione dello stato di terminazione

## ■ Alcuni esempi di attesa processi con wait

Come primo esempio il processo padre genera un solo processo figlio e lo attende con `wait()`.

### ESEMPIO

Scriviamo un primo programma che genera un figlio e utilizziamo la funzione `wait()` in modo che il padre ne attenda la terminazione indicando se la sua esecuzione è terminata in modo corretto.

```
forkwait0.c
1 #include <sys/wait.h>
2 #include <stdio.h>
3 #include <fcntl.h>
4 #include <stdlib.h>
5
6 int main() {
7     int status;
8     int pid = fork();           // creazione del figlio
9     if (pid == 0) {
10         printf("Sono il figlio, il mio pid e': %d. ", getpid());
11         printf("Il mio papi ha pid: %d\n", getppid());
12         exit(69);
13     }
14     else {
15         printf("Sono il padre, il mio pid e': %d. ", getpid());
16         printf("L'exit di mio figlio (%d) e': %d\n", wait(&status), status );
17         return 0;
18     }
19 }
```

Compiliamo il programma e mandiamolo in esecuzione due volte.

```
Paolo@PCwin8 ~/ua1
$ gcc forkwait0.c -o fw0.exe
Paolo@PCwin8 ~/ua1
$ ./fw0
Sono il figlio, il mio pid e': 1292. Il mio papi ha pid: 6704
Sono il padre, il mio pid e': 6704. L'exit di mio figlio (1292) e': 0

Paolo@PCwin8 ~/ua1
$ ./fw0
Sono il figlio, il mio pid e': 7740. Il mio papi ha pid: 4352
Sono il padre, il mio pid e': 4352. L'exit di mio figlio (7740) e': 0

Paolo@PCwin8 ~/ua1
$
```

Possiamo vedere come a ogni esecuzione il **PID** assegnato ai processi è sempre diverso ma il padre termina sempre dopo il figlio.

Il codice può essere migliorato distinguendo e commentando le due possibili situazioni di terminazione, come riportato nel seguente esempio:

```
forkwait1.c
1 #include <sys/wait.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 int main(){
5     int pid, status;
6     pid = fork();
7     if (pid == 0){           // ramo eseguito dal solo processo figlio
8         printf("codice eseguito dal figlio \n");
9         exit(0);
10    }
11    else{                  // ramo eseguito dal solo processo padre
12        pid = wait(&status); // attesa terminazione
13        printf("terminato processo figlio n.%d\n", pid);
14        if ((char)status == 0) // controllo terminazione figlio
15            printf("terminazione volontaria con stato %d\n", status );
16        else
17            printf("terminazione errata con segnale %d\n", (char)status);
18    }
19 }
```

Proviamo ora a togliere l'istruzione `wait()` dalla riga 16 del primo esempio:

```
14 else {
15     printf("Sono il padre, il mio pid e': %d. ", getpid());
16     printf("L'exit di mio figlio e': %d\n", status );
17 }
18 }
```

Compiliamo il programma e mandiamolo in esecuzione due volte.

```
Paolo@PCwin8 ~/ual
$ gcc forkwait1c -o fw0x.exe

Paolo@PCwin8 ~/ual
$ ./fw0x
Sono il padre, il mio pid e': 4048. L'exit di mio figlio e': 0
Sono il figlio, il mio pid e': 6780. Il mio papi ha pid: 4048

Paolo@PCwin8 ~/ual
$ ./fw0x
Sono il padre, il mio pid e': 6608. L'exit di mio figlio e': 0
Sono il figlio, il mio pid e': 6684. Il mio papi ha pid: 6608

Paolo@PCwin8 ~/ual
$
```

Notiamo che il processo padre termina prima del figlio: per effettuare meglio questa osservazione scriviamo un programma che genera più figli.

Come secondo esempio il padre genera più figli e li attende con `wait()`.

**ESEMPIO**

Scriviamo un programma dove il processo padre procede a istanziare quattro figli:

```
forkwait5.c
1 #include <sys/wait.h>
2 #include <stdio.h>
3 #include <fcntl.h>
4 #include <stdlib.h>
5
6 #define NRFIGLI 4
7
8 int main() {
9     int status, pid, x;
10    for (x = 0; x < NRFIGLI; x++) {
11        pid = fork(); // creazione del figlio
12        if (pid == 0) { // nel figlio pid è uguale a 0
13            printf(" Sono il figlio, il mio pid e': %d. ", getpid());
14            printf(" Il mio papi ha pid: %d\n", getppid());
15            exit(0);
16        }
17        else { // nel padre ha valore > 0
18            printf("Sono il padre, il mio pid e': %d. ", getpid());
19            printf("Il pid del figlio corrente e': %d. \n", pid);
20            printf("L'exit di mio figlio (%d) e': %d\n\n", wait(&status), status );
21        }
22    }
23    return 0;
24 }
```

Una sua esecuzione produce il seguente output:

```
Paolo@PCwin8 ~/uai
$ gcc forkwait5.c -o fw5.exe
Paolo@PCwin8 ~/uai
$ ./fw5
Sono il padre, il mio pid e': 4136. Il pid del figlio corrente e': 7868.
Sono il figlio, il mio pid e': 7868. Il mio papi ha pid: 4136
L'exit di mio figlio (7868) e': 0

Sono il padre, il mio pid e': 4136. Il pid del figlio corrente e': 6372.
Sono il figlio, il mio pid e': 6372. Il mio papi ha pid: 4136
L'exit di mio figlio (6372) e': 0

Sono il padre, il mio pid e': 4136. Il pid del figlio corrente e': 6684.
Sono il figlio, il mio pid e': 6684. Il mio papi ha pid: 4136
L'exit di mio figlio (6684) e': 0

Sono il padre, il mio pid e': 4136. Il pid del figlio corrente e': 3944.
Sono il figlio, il mio pid e': 3944. Il mio papi ha pid: 4136
L'exit di mio figlio (3944) e': 0

Paolo@PCwin8 ~/uai
$
```

Il processo padre esegue quattro volte sia la generazione del figlio che il ramo else della istruzione 12, cioè esegue 4 istruzioni di **wait()** e quindi attende uno per uno i quattro figli che ha generato: le coppie padre-figlio vengono eseguite sequenzialmente.

Ora togliamo l'istruzione di riga 20 dove è presente la **wait()**:

```
17 else { // nel padre ha valore > 0
18     printf("Sono il padre, il mio pid e': %d. ", getpid());
19     printf("Il pid del figlio corrente e': %d. \n", pid);
20     // printf("L'exit di mio figlio (%d) e': %d\n\n", wait(&status), status )
21 }
```

Mandiamo in esecuzione il programma ottenendo:

```

Paolo@PCwin8 ~/ua1
$ gcc forkwait5a.c -o fw5.exe
Paolo@PCwin8 ~/ua1
$ ./fw5
Sono il padre, il mio pid e': 7640. Il pid del figlio corrente e': 7184.
  Sono il figlio, il mio pid e': 7184. Il mio papà ha pid: 7640
Sono il padre, il mio pid e': 7640. Il pid del figlio corrente e': 3872.
  Sono il figlio, il mio pid e': 3872. Il mio papà ha pid: 7640
Sono il padre, il mio pid e': 7640. Il pid del figlio corrente e': 4988.
  Sono il figlio, il mio pid e': 4988. Il mio papà ha pid: 7640
Sono il padre, il mio pid e': 7640. Il pid del figlio corrente e': 6756.
  Sono il figlio, il mio pid e': 6756. Il mio papà ha pid: 7640
Paolo@PCwin8 ~/ua1
$ |

```

È immediato verificare come il processo padre termini sempre prima del corrispondente processo figlio.



### Prova adesso!

- Istruzione fork()
- Istruzione wait()
- Istruzione sleep()

Modifica il programma inserendo dei cicli di attesa nel segmento eseguito dai figli utilizzando l'istruzione `sleep(int x)` che sospende "addormenta" per x secondo l'esecuzione del processo che la esegue. Fai in modo che ogni figlio "dorma" per un numero diverso di secondi. Confronta la tua soluzione con quella presente nel file `forkwait5aSol.c`

Verifica infine cosa succede se si dimentica di mettere `exit()` nel ramo eseguito dal figlio (riga 15).

Confronta la tua soluzione con quella presente nel file `forkwait5bSol.c`

### ESEMPIO

Come terzo esempio modifichiamo il codice in modo che sia un processo a generare più figli e attendere solo il primo che termina con una unica `wait()`: per garantirci la casualità nel tempo di elaborazione di ogni figlio introduciamo un ritardo parametrico con valore generato dalla funzione `rand()`.

```

forkwait2.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int pid, pid1, pid2, status, x;
5      srand(time(NULL));
6      pid1 = fork();
7      if(pid1 == 0){           // primo processo figlio
8          x = (rand() % 4) + 1;
9          printf("1) sono il primo processo figlio con pid: %i sleep: %d\n", getpid(), x);
10         sleep(x);
11         exit(1);            // termina primo processo figlio
12     }

```

```

13 } else{
14     pid2 = fork();
15     if (pid2 == 0){           // secondo processo figlio
16         x = (rand() % 2) + 1;
17         printf("2) sono il secondo processo figlio con pid: %i sleep: %d\n", getpid(), x);
18         sleep(x);
19         exit(2);              // termina secondo processo figlio
20     }
21 } else {
22     printf("3) padre in attesa del primo figlio che termina\n");
23     pid = wait(&status);    // attesta terminazione
24     printf("4) per primo termina il figlio con pid: %d\n", pid);
25     return 0;
26 }
27 }
28 }
```

Mandiamo in esecuzione più volte il programma e osserviamo che nella prima esecuzione il figlio che termina per primo è il primo che è stato generato mentre negli altri due casi è il secondo.

```

Paolo@PCwin8 ~/ua1/c_fork_join
$ gcc forkwait2.c -o fw
Paolo@PCwin8 ~/ua1/c_fork_join
$ ./fw
1) sono il primo processo figlio con pid: 7704 sleep: 3
3) padre in attesa del primo figlio che termina
2) sono il secondo processo figlio con pid: 7780 sleep: 1
4) per primo termina il figlio con pid: 7780

Paolo@PCwin8 ~/ua1/c_fork_join
$ ./fw
1) sono il primo processo figlio con pid: 7792 sleep: 2
3) padre in attesa del primo figlio che termina
2) sono il secondo processo figlio con pid: 7788 sleep: 2
4) per primo termina il figlio con pid: 7792

Paolo@PCwin8 ~/ua1/c_fork_join
$ ./fw
1) sono il primo processo figlio con pid: 6016 sleep: 1
3) padre in attesa del primo figlio che termina
2) sono il secondo processo figlio con pid: 6276 sleep: 1
4) per primo termina il figlio con pid: 6016
```

## Funzione `waitpid()`

È possibile sospendere il processo padre in attesa di uno specifico processo utilizzando la funzione:

```
int waitpid(int pid, int *status, int options);
```

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** (identifi-

cativo 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 `setpgid()`. ►

È possibile quindi fare in modo che una processo si sospenda in attesa o di un generico figlio del quale indica il **PID**, oppure di un figlio appartenente a un gruppo da lui definito oppure di un qualunque figlio della sua dinastia.

Nel caso in cui il figlio sul quale viene fatta la `wait()` avesse terminato la sua esecuzione la funzione ritorna immediatamente il valore 0.

Nel caso si verifichi un errore il valore di ritorno è -1 mentre in tutti gli altri casi il valore di ritorno è il **PID** del processo atteso.

### AREA digitale



Valore di OPTIONS in una `waitpid()`

Riscriviamo il programma precedente facendo in modo che il padre termini attendendo il processo più lungo, che realizziamo mediante appositi valori della funzione `sleep()`.

```
forkwait4.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int pid, pid1, pid2, status, x;
5     srand(time(NULL));
6     pid1 = fork();
7     if(pid1 == 0){           // primo processo figlio
8         printf("1) sono il primo processo figlio con pid: %d\n", getpid());
9         sleep(4);
10        exit(1);           // termina primo processo figlio
11    }
12    else{
13        pid2 = fork();
14        if (pid2 == 0){      // secondo processo figlio
15            printf("2) sono il secondo processo figlio con pid: %d\n", getpid());
16            sleep(1);
17            exit(2);          // termina secondo processo figlio
18        }
19        else {
20            printf("3) padre in attesa del figlio piu' lento\n");
21            pid = waitpid(pid1, &status, 0); // attesta terminazione
22            printf("4) finalmente termina il figlio con pid: %d\n", pid);
23            return 0;
24        }
25    }
}
```

Ora qualunque esecuzione ci visualizza una situazione come quella sotto riportata dove il processo padre termina solo dopo la terminazione del processo più lento che è sicuramente il primo figlio dato che ha un ritardo provocato dalla istruzione 9 di `sleep(4)`.

```

Paolo@PCwin8 ~/ua1/c_fork_join
$ gcc forkwait4.c -o fw
Paolo@PCwin8 ~/ua1/c_fork_join
$ ./fw
1) sono il primo processo figlio con pid: 7688
2) sono il secondo processo figlio con pid: 5728
3) padre in attesa del figlio piu' lento
4) finalmente termina il figlio con pid: 7688
Paolo@PCwin8 ~/ua1/c_fork_join
$ |

```



## Prova adesso!

- Istruzione `fork()`
- Istruzione `wait()` e `waitpid()`
- Istruzione `sleep()`

Scrivi un programma dove il processo figlio scrive una frase in un file e il padre ne attende la terminazione della scrittura e successivamente la legge dal file e la visualizza.

Confronta la tua soluzione con quella riportata nel file [forkwait4Sol.c](#)

Quindi modifica il programma aggiungendo tre figli che scrivano in sequenza una frase nel file: il padre visualizza il contenuto del file quando ha terminato il figlio più lento.

Scrivi infine un programma in cui un processo padre P crea due processi figli (prima F1 e poi F2) e ne attende la terminazione in ordine inverso.

Ciascuno dei due figli F1 e F2, a sua volta, crea due processi figli (il figlio F1 crea prima G1, poi H1 e il figlio F2 crea prima G2, poi H2) e ne attende la terminazione nell'ordine in cui li ha creati.

Confronta la tua soluzione con quella riportata nel file [forkwait4aSol.c](#)

## ■ Funzione predefinite per rilevare lo stato

Per interpretare correttamente il valore della variabile `status` è necessario conoscere la rappresentazione di `status`: come precedentemente detto lo standard **POSIX.1** utilizza 16 bit e mette a disposizione delle macro (definite nell'header file `<sys/wait.h>`) per l'analisi dello stato di terminazione.

In particolare

► **bool WIFEXITED(status):** restituisce vero se il processo figlio è **terminato volontariamente**. In questo caso la macro:

`int WEXITSTATUS(status)`

restituisce lo stato di terminazione che passato con la funzione `exit(status)`;

► **bool WIFSIGNALED(status):** restituisce vero se il processo figlio è **terminato involontariamente**.

riamente. In questo caso la macro:

int WTERMSIG(status)

restituisce il numero del segnale che ha causato la terminazione.

Tutti i programmi “dovrebbero” utilizzare queste funzioni per interpretare correttamente i risultati delle elaborazioni: riscriviamo il primo esempio utilizzando queste macro.

```
forkwait6.c
1 #include <sys/wait.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 int main(){
5     int pid, status;
6     pid = fork();
7     if (pid == 0){           // ramo eseguito dal solo processo figlio
8         printf("eseguito dal figlio \n");
9         exit(0);
10    }
11    else{                  // ramo eseguito dal solo processo padre
12        pid = wait(&status); // controllo stato del processo figlio
13        if(WIFEXITED(status))
14            printf("Term. volontaria di %d con stato %d\n", pid, WEXITSTATUS(status));
15        else
16            if(WIFSIGNALED(status))
17                printf("terminazione involontaria per segnale %d\n", WTERMSIG(status));
18    }
19 }
```

Vediamo un esempio più articolato dove creiamo più figli.

### ESEMPIO

Scriviamo un programma dove un processo padre crea un numero N di figli e ciascun figlio scrive 10 volte una lettera dell’alfabeto, rispettivamente il primo figlio scrive le “a”, il secondo le “b”, e via di seguito.

Si richiede di visualizzare un output ordinato, dove le “a” precedono le “b”, le “c” ... ecc.

Naturalmente non ha senso risolvere questo esercizio con le `fork` in quanto dobbiamo rendere sequenziale processi paralleli e quindi, di fatto, rendere nullo il grado di parallelismo.

```
forkwait7.c
1 #include <sys/wait.h>
2 #include <stdio.h>
3 #include <fcntl.h>
4 #include <stdlib.h>
5
6 #define NRFIGLI 4
7 #define VOLTE 10
8
9 int main() {
10     int status, pid, x, y, z;
11     char c;
12     for (x = 0; x < NRFIGLI; x++) {
13         pid = fork();           // creazione del figlio
14         if (pid == 0) {          // nel figlio pid è uguale a 0
15             c = 'a' + x;
```

```

16     for (y = 8; y < VOLTE; y++){
17         printf ("%c",c); fflush(0);
18         sleep(1);
19     }
20     printf ("\n");
21     exit(0);
22 }
23 else // nel padre ha valore <> 0
24     pid = wait(&status); // controllo stato del processo figlio
25     if(WIFEXITED(status))
26         printf("Term. volontaria di %d con stato %d\n", pid, WEXITSTATUS(status));
27     else
28         if(WIFSIGNALED(status))
29             printf("terminazione involontaria per segnale %d\n", WTERMSIG(status));
30 }
31 }
```



## Prova adesso!

- Istruzione wait
- Macro di controllo dello status

Scrivi un programma che genera una **gerarchia di processi** ricevendo dalla linea di comando tre parametri

<flag> <#\_figli> <#\_nipoti >

Dove rispettivamente:

**<flag>**: permette di indicare il tipo di concorrenza:

- ▶ 0: figli in parallelo;
- ▶ 1: esecuzione figli in sequenza;

**<#\_figli>**: numero di figli di primo livello nella gerarchia;

**<#\_nipoti >**: numero di nipoti che ciascun figlio deve fare.

Esempi di parametri

- ▶ 0 1 2: parallelo , un figlio con due nipoti
- ▶ 1 2 2 4: seriale, due figli rispettivamente uno con 2 nipoti e uno con 4
- ▶ 1 3 1 2 3: seriale, tre figli rispettivamente uno con 1, 2 e 3 nipoti

```

Paolo@PCwin8 ~/ua1/c_fork_join
$ ./f 0 1 2
PADRE: Creato figlio (pid=4060)
Figlio 4060: Creato nipote (pid=8928)
Nipote 8928, figlio di 4060
PADRE: terminazione volontaria del figlio 8928 con stato 0
PADRE: terminazione volontaria del figlio 4060 con stato 0
PADRE: Creato figlio (pid=5628)
Figlio 5628: Creato nipote (pid=7384)
Nipote 7384, figlio di 5628
PADRE: terminazione volontaria del Figlio 7384 con stato 0
Figlio 5628: Creato nipote (pid=8832)
Nipote 8832, figlio di 5628
PADRE: terminazione volontaria del Figlio 8832 con stato 0
PADRE: terminazione volontaria del Figlio 5628 con stato 0
Paolo@PCwin8 ~/ua1/c_fork_join
$ |
```

Confronta la tua soluzione con quella riportata nel file **forkwait7Sol.c**

# ESERCITAZIONI DI LABORATORIO 6

## FORK-JOIN E COBEGIN-COEND



### Info

I codici sorgenti sono nel file [C\\_fork\\_join.rar](#) scaricabile dalla cartella materiali nella sezione del sito [www.hoepliscuola.it](http://www.hoepliscuola.it) riservata al presente volume.

### ■ Il costrutto fork-join

Utilizzando le funzioni `fork()` e `wait()` realizziamo il costrutto **fork-join** per l'esecuzione di calcoli paralleli dove il processo figlio passa il risultato al processo padre per generare il risultato finale.

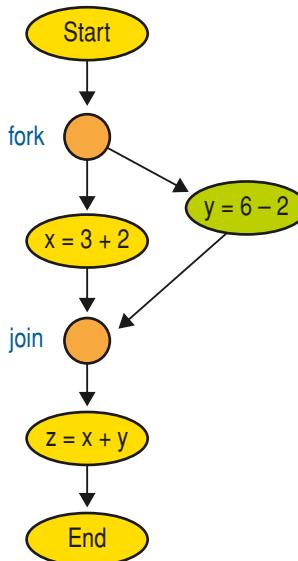
#### ESEMPIO

Risolviamo per esempio 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:



Una pseudocodifica del programma è la seguente:

```

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

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

```

Separiamo il codice che effettua i calcoli matematici dal codice che controlla l'evoluzione dei processi: una elaborazione viene fatta dal processo figlio mentre due elaborazioni sono effettuate dal processo padre.

```

forkjoin1.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int calcoli_figiol(){
5     int k;
6     printf("1.1 elab. parallela processo figlio \n");
7     k = 3 + 2;
8     return k;
9 }
10
11 int calcoli_padre1(){
12     int k;
13     printf("1 elab. parallela processo padre \n");
14     k = 6 - 2;
15     return k;
16 }
17
18 int calcoli_padre2(int a, int b){
19     int k;
20     printf("2 elab. finale padre \n");
21     k = (a + b);
22     return k;
23 }

```

Ricordiamo che è possibile leggere il valore di ritorno **ret\_value** della funzione

```
int wait(*ret_value)
```

mediante la funzione

```
int WEXITSTATUS(ret_value)
```

e sfruttiamo questa particolarità per effettuare il “ritorno” del valore calcolato dalle funzioni al processo padre, come possiamo vedere nella istruzione 37, se tale valore è inferiore al numero 255.

```

24
25  int main( ){
26      int x, y, z , retv;
27      pid_t pid;
28      pid = fork();           // inizio elaborazione parallela
29      if (pid == 0){
30          x = calcoli_figlio(); // esecuzione parallela calcoli figlio
31          exit(x);            // termina processo figlio
32      }
33      else{
34          y = calcoli_padre(); // esecuzione parallela calcoli padre
35      }
36      printf(.. join: padre aspetta \n");
37      wait(&retv);           // join : il padre aspetta il figlio
38      x = WEXITSTATUS(retv); // prende il risultato del figlio
39      z = calcoli_padre2(x, y); // esegue gli ultimi calcoli
40      printf("-> risultato finale z = %d", z );
41  }

```

Mandiamo in esecuzione il programma e otteniamo il seguente output:

```

~/ua1/c_fork_join
Paolo@PCwin8 ~/ua1/c_fork_join
$ ./fj
1 elab. parallela processo padre
.. join: padre aspetta
1.1 elab. parallela processo figlio
2 elab. finale padre
-> risultato finale z = 9
Paolo@PCwin8 ~/ua1/c_fork_join
$ |

```

Utilizzare il parametro di **status** per “comunicare” con il processo padre non è sicuramente una soluzione percorribile nel caso che si voglia trasferire il risultato di elaborazioni: anche solo un numero negativo manderebbe in difficoltà questa soluzione.

Gli esercizi presenti in questa lezione sono quindi di carattere esclusivamente teorico: vedremo nel prossima esercitazione come comunicare tra **processi** mediante aree di **memoria condivisa**.



## Prova adesso!

- Costrutto fork-join

Dopo aver realizzato il diagramma delle precedenze, scrivi i programmi che implementando il costrutto **fork-join** eseguono le seguenti operazioni con il massimo grado di parallelismo.

$$5 * [(2 + 4) * (7 + 3)] - 10$$

$$(3 + 2) * (5 - 7) + (8 - 3)$$

$$(3 + x) - (5 - y) * (7 * y + 3) \quad // x viene letto nel padre mentre y e z nel main()$$

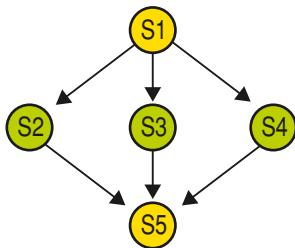
Confronta il tuo codice con quello presente nel file **forkjoin1Sol.c**.

## ■ 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

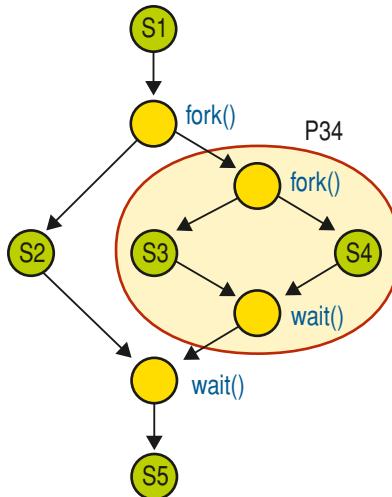


descritta con la seguente pseudocodifica

```

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:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main( ){
5      int status;
6      pid_t pid_figlio;
7      printf ("S1 (padre1) - pid = %d\n", getpid());
8      printf( " fork esterna - \n" );
9      if ((pid_figlio = fork()) == -1) // prima fork
10         printf( "fork non riuscita !" );
11     else
12     {
13         if (pid_figlio == 0)
14             printf(" figlio1: pid = %d, padre pid = %d\n", getpid(), getppid());
15         ramo34(); // esecuzione fork processi 3 e 4 (P34)
16     }
17 }
```

Il ramo di destra ora deve eseguire la seconda fork in modo che venga eseguito il segmento S3 in parallelo col segmento S4:

```
26  ramo34(){
27      int status;
28      pid_t pid4;
29      printf( " fork - processi P34 -> creazione P4 \n" );
30      if (( pid4 = fork() ) == -1 )
31          printf( " fork non riuscita !" );
32      if (pid4 == 0){
33          printf(" S4 (figlio2)- pid = %d, padre pid = %d\n", getpid(), getppid() );
34      }
35      else{
36          printf(" S3 (padre2) - pid = %d, padre pid = %d\n", getpid(), getppid() );
37          printf(" wait fine figlio P4 (padre2 join figlio2) \n" );
38          waitpid(pid4, &status , 0); // attesa fork interna
39          printf(" fine attesa P4 \n" );
40          printf(" fine ramo P34 \n" );
41      }
42      exit( 0 );
43 }
```

Contemporaneamente viene eseguito il ramo di sinistra, cioè il segmento S2, e alla fine il **main()** si mette in attesa della terminazione del ramo di destra, per poi poter eseguire l'ultima parte di codice, il segmento S5.

```
15  }
16  else{ // esecuzione ramo padre
17      printf(" S2 (padre1) - pid = %d\n",getpid());
18      printf(" wait fine ramo P34 (padre1 join figlio1) \n" );
19      waitpid(pid_figlio, &status , 0); // attesa prima fork
20      printf(" fine attesa P34 \n" );
21      printf("S5 (padre1)- pid = %d\n",getpid());
22      exit( 0 );
23  }
24 }
```

Abbiamo indentato manualmente l'output in modo da individuare sullo schermo le diverse istruzioni di **join** e **wait**:

```

Paolo@PCwin8 ~/ua1/c_fork_join
$ gcc cobegin1.c -o cb

Paolo@PCwin8 ~/ua1/c_fork_join
$ ./cb
S1 (padre1) - pid = 6988
fork esterna -
S2 (padre1) - pid = 6988
wait fine ramo P34 (padre1 join figlio1)
figlio1: pid = 7528, padre pid = 6988
fork - processi P34 -> creazione P4
S3 (padre2) - pid = 7528, padre pid = 6988
wait fine figlio P4 (padre2 join figlio2)
S4 (figlio2)- pid = 3584, padre pid = 7528
fine attesa P4
fine ramo P34
fine attesa P34
S5 (padre1)- pid = 6988

Paolo@PCwin8 ~/ua1/c_fork_join
$ |

```

Non è agevole scambiare informazioni tra padre e figlio in quanto i processi non condividono variabili: al momento della **fork** tutte le variabili del padre vengono duplicate nel figlio e se effettuiamo il passaggio dei parametri per indirizzo spesso ci troviamo di fronte a situazioni complesse.

Nella prossima lezione vedremo un meccanismo più semplice ed efficace per far comunicare tra loro i processi.



## Prova adesso!

- Costrutto cobegin-coend

Dopo aver realizzato il diagramma delle precedenze, scrivi i programmi che implementando il costrutto **cobegin-coend** eseguono le seguenti operazioni:

$$5 * [(2 + 4) * (7 + 3)] - 10$$

$$(3 + 2) * (5 - 2) * (8 - 3)$$

$$(2 + x) * (3 + y) * (7 * z + 4) - 10 \quad // x, y e z vengono letti da input nel main()$$

Confronta il tuo codice con quello presente nel file **cobegin1Sol.c**



# ESERCITAZIONI DI LABORATORIO 7

## 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](http://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 ad 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:

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

Dove:

- **ptID**: è il puntatore alla variabile che contiene l’identificativo del **thread** che viene creato, il **thread\_ID (tID)**;
- **att**: è il puntatore all’eventuale vettore (o alla variabile) contenente i parametri (od il parametro) da passare al **thread** creato (per esempio la priorità); noi utilizzeremo sempre gli attributi di default e quindi indicheremo nella chiamata **NULL**;

- **routine:** è il puntatore alla funzione che contiene il codice che il **thread** deve eseguire; tale funzione deve avere il seguente prototipo:

```
void *codice_thread (void *argomento);
```

- **arg:** è il puntatore all'eventuale vettore (o alla variabile) contenente i parametri (od il parametro) da passare al **thread** creato e può essere posto a **NULL** nel caso di assenza di parametri.

**void \*** permette di passare alla funzione argomenti di natura e struttura diversi: nella funzione eseguita dal **thread** è possibile, tramite **cast**, tradurre il puntatore generico in un puntatore ad una struttura dati che contiene tutti i parametri necessari per quella specifica applicazione.

La funzione restituisce 0 in caso che vada a buon fine la creazione di un nuovo **thread** e il **TID** del **thread** appena creato è salvato all'interno di **pthread\_t**, altrimenti ritorna un codice di errore diverso da zero secondo le convenzioni di **<sys/errno.h>**.

Come sappiamo il **thread** condivide con il **processo invocante** le seguenti aree:

- codice;
- dati;
- mappe di memoria.

In seguito vedremo come utilizzare questa condivisione per scambiare dati tra processo e **thread** e tra i singoli **thread**.

Vediamo un primo esempio dove effettuiamo la creazione di un **thread** con **pthread\_create()**:

```
creazione.c
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 void *codice_thread(void * arg) {      // codice che esegue il thread
6     pid_t pid;
7     pid = getpid();          // ritorna l'identificatore del processo pID
8     pthread_t tid;
9     tid = pthread_self();    // ritorna identificatore del thread (il valore di ptid)
10    printf("Sono il thread %i del processo %i\n", tid, pid);
11 }
12
13 int main (){
14     pthread_t ptid;        // identificatore thread alla creazione
15     printf ("il pid del main e' %d\n", (int) getpid ());
16     pthread_create (&ptid, NULL, &codice_thread, NULL);
17     return (void*)0;        // return 0;
18 }
```

Compiliamo e mandiamo in esecuzione il programma, che visualizza una situazione simile alla seguente:

```
Paolo@PCwin8 ~/u1/c_pthread
$ gcc creazione.c -o crea
Paolo@PCwin8 ~/u1/c_pthread
$ ./crea
il pid del main e' 6672
Paolo@PCwin8 ~/u1/c_pthread
$
```

Non visualizzando l'output della riga 10 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 la sua terminazione comporta la terminazione anche del figlio “prima che nasca”.

Per essere sicuri che entrambi vengano eseguiti è necessario introdurre un ritardo nel padre in modo da “permettere” la nascita del figlio stesso: modifichiamo il codice, come di seguito riportato, inserendo un secondo di ritardo nel processo padre.

```

13 int main (){
14     pthread_t miothread;
15     printf ("il pid del main e' %d\n", (int) getpid ());
16     if (pthread_create (&miothread, NULL, &codice_thread, NULL)==0) // tutto ok
17         sleep(1);           // ritardo per garantire l'esecuzione del thread
18     else
19         printf ("errore nella creazione del thread! " );
20     return 0;
21 }
```

In esecuzione ora otteniamo il seguente output:

```

Paolo@PCwin8 ~/ua1/c_pthread
$ ./crea
il pid del main e' 6672
Sono il thread 296080 del processo 6672
Paolo@PCwin8 ~/ua1/c_pthread
$
```

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

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**, come fatto con l'istruzione 16 del listato precedente.



## Prova adesso!

- Creazione di thread

Scrivi un programma che crea tre thread che, rispettivamente, visualizzano sullo schermo ciascuno il suono di una campana: DIN , DON oppure DAN.

Manda più volte in esecuzione il programma osservando l'output dopo aver introdotto un ritardo casuale in ciascun **thread**.

Confronta la tua soluzione con quella riportata nel file **campaneSol.c**

## Terminazione di thread: pthread\_exit

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

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

```
return();
```

- Ⓑ 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**.

È anche possibile terminare il **thread** utilizzando la classica funzione **exit()**.

Si consiglia di utilizzare sempre la chiamata di **pthread\_exit()** oppure **return()** evitando l'uso di **exit()** per la terminazione regolare di un **processo/thread** riservando questa per i casi di uscita forzata e immediata (errore irrecuperabile).

Il valore di ritorno **retval** della funzione **pthread\_exit()** è anch'esso di tipo **void \*** in modo da permettere di passare alla funzione chiamante argomenti di natura e struttura diversi: ritornando un puntatore generico il processo chiamante effettuerà tramite cast la conversione in un puntatore a una struttura dati specifica per quella applicazione nella quale il thread ha inserito i risultati che vuole passare al chiamante.

Abbiamo due possibili strade per “produrre” il parametro di ritorno, a seconda della dimensione del dato:

- si può convertire direttamente il dato presente in **void \*** se il valore da restituire è un intero oppure è un tipo più piccolo della dimensione dell'intero;
- si deve convertire il puntatore al dato **void \*** se il valore da restituire ha una dimensione maggiore di quella di un intero.

È da preferirsi quindi la soluzione che prevede sempre di effettuare la conversione tra puntatori in quanto è più elegante e più portabile: inoltre non dipende in alcun modo dalla dimensione dei dati dei vari tipi in caso di esecuzione su piattaforme hardware diverse.

Se dobbiamo restituire un valore costante intero possiamo utilizzare la seguente notazione:

```
11 |
12 |   pthread_exit( (int*) 0); // ritorna un numero intero
13 }
```

dove ritorniamo il numero 0 al programma che lo ha generato: vedremo in seguito come leggere tale parametro utilizzando la funzione **join**.

Dal momento che la funzione **void pthread\_exit()** non ha nessun parametro di ritorno, il codice successivo alla chiamata di questa funzione è “codice morto” che non verrà mai eseguito.

## ■ Cancellazione di un thread

È possibile cancellare un **thread** prima che termini la sua esecuzione mediante il comando:

```
int pthread_cancel(pthread_t tid);
```

richiede come parametro in ingresso il **tid** del **thread** che deve essere terminato e restituisce come parametro in uscita:

- 0 se OK;
- un codice d'errore se l'istruzione non va a buon fine.

## ■ 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);
```

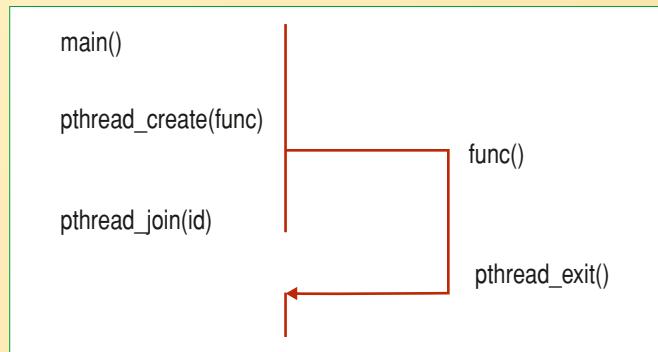
Dove:

- **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.

Come valore di ritorno **int** la funzione restituisce 0 in caso di successo, altrimenti un codice di errore diverso da 0, secondo le convenzioni di **<sys/errno.h>**.

La funzione **pthread\_join()** è equivalente alla **waitpid()** dei processi fatta con la **fork()**; ha un comportamento come quello riportato in figura:



Riscriviamo ora il codice dell'esempio **campaneSol.c** facendo in modo di ottenere la sequenza ordinata del suono delle campane utilizzando la funzione **pthread\_join()**.

Il codice dei tre figli è per esempio il seguente:

```
campaneJoin.c
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 void* thread1 (void* arg){
6     sleep(rand() % 1);
7     printf ("DIN ");
8     fflush(stdout);
9 }
10 void* thread2 (void* arg){
11     sleep(rand() % 2);
12     printf ("DON ");
13     fflush(stdout);
14 }
15 void* thread3 (void* arg){
16     sleep(rand() % 3);
17     printf ("DAN ");
18     fflush(stdout);
19 }
```

Il codice del padre è molto semplice: crea i tre figli e si mette in attesa della loro terminazione.

```
18 int main (){
19     srand(time(NULL));
20     pthread_t miothread1, miothread2, miothread3 ;
21     printf ("il pid del main e' %d\n", (int) getpid ());
22     pthread_create (&miothread1, NULL, &thread1, NULL);
23     pthread_create (&miothread2, NULL, &thread2, NULL);
24     pthread_create (&miothread3, NULL, &thread3, NULL);
25     pthread_join (miothread1, NULL);
26     pthread_join (miothread2, NULL);
27     pthread_join (miothread3, NULL);
28     return 0;
29 }
```

Mandiamo in esecuzione più volte il programma e otteniamo risultati diversi a ogni sua istanza.

```
Paolo@PCwin8 ~/uai/c_pthread
$ ./pt
il pid del main e' 2112
DAN DON DIN
Paolo@PCwin8 ~/uai/c_pthread
$ ./pt
il pid del main e' 6980
DON DIN DAN
Paolo@PCwin8 ~/uai/c_pthread
$ ./pt
il pid del main e' 7340
DON DAN DIN
Paolo@PCwin8 ~/uai/c_pthread
$ |
```

Per ottenere un risultato deterministico è necessario introdurre delle rettifiche che lasciamo come esercizio per il lettore.

Riportiamo in una tabella riepilogativa le similitudini esistenti tra processi e **thread**.

Processi	Thread
fork	pthread_create
exit	pthread_exit
waitpid	pthread_join
kill	pthread_kill
getpid	pthread_self



◀ Quando compiliamo un programma che utilizza i thread in ambiente Linux nativo, dobbiamo necessariamente richiamare la libreria **pthread** all'interno della linea di comando come riportato di seguito: ►

```
C_pthread : bash - Konsole
File Edit View Bookmarks Settings Help
utente@AspireXC705-Linux: ~ > cd sviluppo
utente@AspireXC705-Linux: ~/sviluppo > cd ual
utente@AspireXC705-Linux: ~/sviluppo/ual > cd C_pthread
utente@AspireXC705-Linux: ~/sviluppo/ual/C_pthread > gcc -pthread creazione.c -o crea
utente@AspireXC705-Linux: ~/sviluppo/ual/C_pthread > ./crea
il pid del main e' 3758
Sono il thread 429135616 del processo 3758
C_pthread : bash
```



## Prova adesso!

- Costrutto fork-join
- Costrutto cobegin-coend

Realizza mediante i thread gli stessi programmi che implementavano il costrutto fork-join e cobegin-coend fatti per i processi.

In particolare:

- A utilizzando il costrutto **fork-join** realizzare le seguenti operazioni con il massimo grado di parallelismo:

$$5 * [(2 + 4) * (7 + 3)] - 10 \\ (3 + x) - (2 + y) * (7 * y + 3) \quad // x e y vengono letti da input nel main()$$

- B utilizzando il costrutto **cobegin-coend** realizzare le seguenti operazioni con il massimo grado di parallelismo:

$$(3 + 2) * (5 - 2) * (8 - 3) \\ (2 + x) * (3 + x) * (7 * y + 4) - 10 \quad // x e y vengono letti da input nel main()$$

# ESERCITAZIONI DI LABORATORIO 8

## THREAD E PARAMETRI

### ■ Passare parametri a un thread

Abbiamo due possibili alternative per scambiare dati tra il programma principale e i **thread** che da esso vengono creati:

- Ⓐ **passaggio diretto**
- Ⓑ **passaggio indiretto in memoria condivisa**

Col **passaggio diretto** vengono sfruttate le possibilità offerte rispettivamente dalla funzione:

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

utilizzando l'argomento **void \*arg** per passare un valore dal “creatore” al **thread**, e dalla funzione:

```
int pthread_exit(void *status)
```

per ritornare un valore dal **thread** al “creatore” utilizzando il parametro **void \*status**.

Ricordiamo che l'argomento e il valore di ritorno sono puntatori generici, quindi di tipo **void**, per offrire la massima flessibilità di scambio variabili.

In tal modo nella funzione eseguita dal **thread** è possibile:

- trasformare in ingresso tramite **casting** il puntatore generico in un puntatore a una specifica struttura dati (che contiene tutti i parametri richiesti dalla specifica applicazione);
- ritornare un puntatore generico il quale, sempre tramite **casting**, sarà convertito in un puntatore a una struttura dati che contiene tutti i risultati che il **thread** vuole trasmettere al processo chiamante.

### ■ Parametro passato da principale a thread

Come primo esempio passiamo al momento della creazione del **thread** un numero intero come ultimo parametro mediante la variabile **ingresso**:

```
30 int main() {
31     pthread_t tid;
32     int ingresso; // utilizzata come variabile di ingresso condivisa
33     void *ritorno; // utilizzata come variabile di ritorno condivisa
34     ingresso = 1;
35     pthread_create(&tid, NULL, thread, (void*) &ingresso); // &ingresso al thread
```

La corretta istruzione per leggere il valore del parametro è la seguente:

```
7 void *thread(void *arg) {
8     int dato, *punta_dato;
9     // 1) lettura diretta del valore passato come parametro
10    dato = *(int*) arg;
11 }
```

che può essere fatta anche in due istruzioni convertendo dapprima il puntatore e successivamente dereferenziandolo:

```
12 // 2) lettura in due passaggi
13 punta_dato = (int*) arg;           // cast da void* ad int*
14 dato = *punta_dato;              // dereferenziazione e lettura valore
15
```

Con la stessa procedura possiamo leggere un parametro di qualsiasi formato: nel seguente esempio leggiamo una stringa.

```
5 void *thread1(void *arg) {
6     char *nome;
7     nome = (char*) arg;           // scrivo nell'argomento arg
8 }
```

che viene passata mediante la seguente istruzione di creazione:

```
18 char *ingresso = "Thread 1";
19 pthread_create(&tid, NULL, thread1, (void*) ingresso); // &ingresso al thread
20
```



## Prova adesso!

- Passaggi parametri a thread

Scrivi un programma che crea un **thread** per ciascuno dei sette nani e ne visualizza il nome sullo schermo utilizzando il seguente segmento di codice:

```
31     nome[0] = "Cucciolo";
32     nome[1] = "Pisolo";
33     nome[2] = "Eolo";
34     nome[3] = "Dotto";
35     nome[4] = "Mammolo";
36     nome[5] = "Gongolo";
37     nome[6] = "Brontolo";
38     for (t = 0; t < NUM_THREADS; t++) {
39         printf("Creazione thread %d\n", t);
40         rc = pthread_create(&threads[t], NULL, codice_thread, (void *) &t);
41 }
```

Controlla la tua soluzione con quella presente nel file [naniSol.c](#)

Quindi modifica il programma [campane.c](#) passando ai singoli **thread** il nome della campana che deve suonare.

Controlla la tua soluzione con quella presente nel file [campaneParametroSol.c](#)

## ■ Parametro di ritorno da thread a chiamante

Passare dati da un **thread** al chiamante richiede attenzione in quanto è necessario

- prenderne l'indirizzo con l'operatore **&**;
- convertire l'indirizzo in **void\*** col casting (**void\***);
- restituire tale valore passandolo con **pthread\_exit()** o con l'istruzione **return()**.

### ESEMPIO

Scriviamo come esempio un programma dove il **thread** effettua il lancio di un dato e ritor-  
na il valore generato mediante la funzione **pthread\_exit()**.

dadoGlobale.c

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int parametroOUT;
6  void* codice(void *arg){                                // thread che genera un dado
7      srand(time(NULL));
8      parametroOUT = (rand() % 6) +1;
9      pthread_exit( (void*) &parametroOUT );
10 }
11
12 int main () {
13     int dadoestratto, *risultato;
14     pthread_t t1;
15     pthread_create(&t1, NULL, codice, NULL);
16     pthread_join(t1, (void*) &risultato);
17     printf("dato estratto: %d\n", *risultato);
18     return 0;
19 }
```

Alcune esecuzioni danno il seguente risultato:

```

E ~ /u1/c_pthread
Paolo@PCwin8 ~/u1/c_pthread
$ gcc dadoGlobale.c -o dg
Paolo@PCwin8 ~/u1/c_pthread
$ ./dg
dato estratto: 2

Paolo@PCwin8 ~/u1/c_pthread
$ ./dg
dato estratto: 5

Paolo@PCwin8 ~/u1/c_pthread
$ ./dg
dato estratto: 4

```

Il dato da restituire non deve essere contenuto in una variabile automatica della funzione in esecuzione nel **thread** dato che le variabili automatiche risiedono sullo stack del **thread** che al termine della sua esecuzione viene rilasciato e quindi l'indirizzo restituito punta a una zona di memoria non più esistente.



## Prova adesso!

- Utilizzo funzione pthread\_exit()
- Utilizzo funzione pthread\_join()

Sposta l'istruzione 5 di dichiarazione della variabile globale all'interno del codice del **thread**, come riportato in figura.

```

4
5 void* codice(void *arg){
6     int parametroOUT;
    
```

// thread che genera un dato  
// dichiarata come locale (errore)

Manda in esecuzione il programma e osserva i risultati: come puoi giustificarli? Come puoi modificare ulteriormente il codice per ottenere "almeno" un valore di ritorno al programma chiamante?

Confronta la tua soluzione con quelle riportata nel file **dadoErratoSol.c**

## ■ Passaggio dati mediante condivisione di memoria

Abbiamo visto come sia abbastanza complesso ricevere da un **thread** i risultati della sua elaborazione e nel caso ci fossero più valori di tipo diverso si rende necessaria la definizione di una struttura di dati: inoltre abbiamo visto che affinché si possa riportare l'indirizzo la struttura deve essere definita al di fuori del **thread**.

Nel programma precedente il **thread codice** passa come parametro di ritorno al **main** il puntatore alla variabile che però deve essere dichiarata nel **main** altrimenti non esiste al termine della esecuzione del **thread**: in altre parole passiamo al **main** il puntatore di una variabile che è definita nel **main**!

A questo punto possiamo semplificare queste operazioni e sfruttare direttamente lo stesso spazio di indirizzamento comune che i **thread** condividono senza passare alcuna variabile, né in ingresso e né in uscita, cioè tutti i **thread** vedono le stesse **variabili globali** e se uno di essi ne modifica una tale modifica è vista anche dall'altro **thread**.

In tutti i nostri esempi l'accesso alla **variabile globale** non viene regolamentato da alcun meccanismo di gestione della concorrenza: questo potrebbe provocare anomalie e malfunzionamenti, come discusso nella prossima **unità didattica**.

Scriviamo un esempio riepilogativo dove una **variabile globale** viene modificata da un **thread** in maniera causale elaborando un dato letto come parametro in ingresso e ritornando 0 oppure 1 in base al valore di questa variabile.

```
parametri.c
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <pthread.h>
4 int globale = 30;
5
6 void *thread(void *arg) {
7     int dato;
8     // elaborazione parametro in ingresso
9     dato = *(int*) arg + 20;           // scrivo nell'argomento arg
10
11    // introduco casualità del risultato
12    srand(time(NULL));
13    globale = globale * (rand() % 7) + dato;
14
15    // alternativa di terminazione condizionata
16    if (globale < 100 )
17        pthread_exit( (int*) 0);      // ritorno numero 0
18    else
19        pthread_exit( (int*) 1);      // ritorno numero 1
20 }
```

Il `main()` passa il parametro `ingresso` alla creazione del `thread` e aspetta la sua terminazione leggendo il parametro `ritorno`: quindi visualizza i loro valori.

```
22 int main() {
23     pthread_t tid;
24     int ingresso;    // utilizzata come variabile di ingresso condivisa
25     void *ritorno;   // utilizzata come variabile di ritorno condivisa
26     ingresso = 10;
27     pthread_create(&tid, NULL, thread, (void*) &ingresso); // &ingresso al thread
28     if ( pthread_join(tid, &ritorno) == 0 ) {                // legge valore ritorno
29         printf(" valore di globale : %d\n", globale);
30         printf(" valore di ingresso: %d\n", ingresso);
31         printf(" uscita di ritorno : %d\n", ritorno);
32         exit(0);
33     }
34     else
35         printf ("errore join del thread! .");
36 }
37 }
```

Due possibili esecuzioni danno il seguente risultato:

```
Paolo@PCwin8 ~/uai/c_pthread
$ ./parametri
valore di globale : 60
valore di ingresso: 10
uscita di ritorno : 0

Paolo@PCwin8 ~/uai/c_pthread
$ ./parametri
valore di globale : 120
valore di ingresso: 10
uscita di ritorno : 1

Paolo@PCwin8 ~/uai/c_pthread
$
```



## Prova adesso!

- Condivisione di variabili globali

Scrivi un programma che calcola la somma dei primi  $2^{N-1}$  numeri interi con N letto da input creando N **thread**.

Confronta la tua soluzione con quella presente nel file [potenzaSol.c](#)

Scrivi un programma che riceve in ingresso un numero N intero da linea di comando, genera N **thread** e ne aspetta la terminazione: ogni **thread** genera un numero casuale X, attende X secondi e quindi lo incrementa a una variabile globale totale.

Al termine della elaborazione viene visualizzato il valore di questa variabile.

Confronta la tua soluzione con quella presente nel file [addizioniSol.c](#)

## ■ Errato utilizzo delle variabili globali

Scriviamo ora un semplice programma che crea sette **thread** dove ciascuno di essi visualizza sullo schermo il nome di uno dei sette nani (Cucciolo, Pisolo, Eolo, Dotto, Mammolo, Gongolo, Brontolo) definendo **globale** il vettore con i nomi dei nani e passando a ciascun **thread** il rispettivo identificatore **tid**.

Invece di generare sette valori distinti di **tid**, si usa una variabile comune e la si condivide fra **thread** passandola come parametro: tale parametro viene dereferenziato a una variabile intera usata dal **thread** per leggere e visualizzare il proprio nome da una array **nome[ ]**.

```
nani.c
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <stdlib.h>           // per utilizzare costanti predefinite
5
6 #define NUM_THREADS 7          // nr dei thread da far partire
7 char *nome[NUM_THREADS];    // vettore contenente i nomi dei nani
8
9 void *cod_thread(void *tid){
10     int tid_int;
11     tid_int = *(int*) tid;      // leggo parametro da puntatore void
12     // generazione ritardo casuale
13     srand(time(NULL));
14     sleep(rand() % 2);
15     printf(" Ciao da %d: %s \n", tid_int, nome[tid_int]); // stampa nome
16     pthread_exit(NULL);        // uscita del thread senza valori di ritorno
17 }
```

Il main() riempie con i nomi dei nani il vettore **nome[ ]**, quindi crea i sette **thread** passando a ciascuno il rispettivo **pid**.

```

19 int main(int argc, char *argv[]){
20     pthread_t threads[NUM_THREADS]; // vettore identificare dei thread
21     int tid, rc;
22     nome[0] = "Cucciolo";
23     nome[1] = "Pisolo";
24     nome[2] = "Eolo";
25     nome[3] = "Dotto";
26     nome[4] = "Mammolo";
27     nome[5] = "Gongolo";
28     nome[6] = "Brontolo";
29
30     for (tid = 0; tid < NUM_THREADS; tid++) {
31         printf("Creazione thread %d\n", tid);
32         rc = pthread_create(&threads[tid], NULL, cod_thread, (void *) &tid);
33         // sleep(1); // necessaria: la velocità della macchina crea problemi sul valore di t
34         if (rc) {
35             printf ("ERRORE: il codice di errore di ritorno da pthread_create() e': %d\n", rc);
36             exit(EXIT_FAILURE);
37         }
38     }
39     pthread_exit(NULL); // terminazione corretta del programma
40 }
41

```

Poiché `pthread_create()` è asincrona, il processo principale fa in tempo a impostare `tid = 7`, che è la condizione di uscita della istruzione 30, prima che siano partiti tutti i `thread` e quindi come risultato avremo che alcuni `thread` (uno sicuramente!) stampa il contenuto della cella `nome[7]` che, nella migliore delle ipotesi, è `NULL`.

```

Paolo@PCwin8 ~/ua1/c_pthread
$ ./nani
Creazione thread 0
Creazione thread 1
Ciao da 1: Pisolo
Creazione thread 2
Ciao da 2: Eolo
Creazione thread 3
Ciao da 3: Dotto
Creazione thread 4
Ciao da 4: Mammolo
Creazione thread 5
Ciao da 5: Gongolo
Creazione thread 6
Ciao da 6: Brontolo
Ciao da 7: (null)
Paolo@PCwin8 ~/ua1/c_pthread
$ 

```

Per eliminare il malfunzionamento è necessario introdurre un ritardo nel programma principale (istruzione 33) in modo da permettere al figlio di leggere correttamente il proprio nome.



## Prova adesso!

- Condivisione di variabili globali

Scrivi un programma che crea otto **thread** e, rispettivamente, ciascuno visualizza sullo schermo il saluto in una lingua diversa senza usare variabili globali come contatori ma definendo globale il vettore con i messaggi di saluto.

```

32  /* I messaggi nelle otto lingue */
33  messaggio[0] = "English: Hello World!";
34  messaggio[1] = "French: Bonjour, le monde!";
35  messaggio[2] = "Spanish: Hola al mundo";
36  messaggio[3] = "Italiano: Ciao mondo!";
37  messaggio[4] = "German: Guten Tag, Welt!";
38  messaggio[5] = "Russian: Zdravstvyyte, mir!";
39  messaggio[6] = "Japan: Sekai e konnichiwa!";
40  messaggio[7] = "Latin: Orbis, te saluto!";
...
```

Manda più volte in esecuzione il programma osservando l'output dopo aver introdotto un ritardo casuale in ciascun **thread** e modifica il programma per renderlo immune da errori. Confronta la tua soluzione con quella riportata nel file **multilinguaSol.c**.

### AREA digitale



Un ulteriore esercizio errato sulla memoria condivisa

### AREA digitale



Esercizi per il recupero / Esercizi per il rinforzo

# ESERCITAZIONI DI LABORATORIO 10

## I THREAD IN JAVA: CONCETTI BASE



### Info

I codici sorgenti sono nel file **java\_thread.rar** scaricabile dalla cartella **materiali** nella sezione del sito [www.hoepiscuola.it](http://www.hoepiscuola.it) riservata al presente volume.

### ■ I thread 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 il primo 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
<code>int MIN_PRIORITY, NORM_PRIORITY, MAX_PRIORITY</code>	<code>Thread()</code> <code>Thread(Runnable target)</code> <code>Thread(String name)</code> <code>Thread(ThreadGroup name, Runnable target)</code> <code>Thread(ThreadGroup group, String name)</code> <code>Thread(Runnable target, String name)</code> <code>Thread(ThreadGroup group, Runnable target, String name)</code> <code>Thread(ThreadGroup group, Runnable target, String name, long stackSize)</code>	<code>int activeCount()</code> <code>Thread currentThread()</code> <code>void dumpStack()</code> <code>int enumerate(Thread tarray[])</code> <code>boolean holdsLock(Object obj[])</code> <code>boolean interrupted()</code> <code>void sleep(long millis)</code> <code>void sleep(long millis, int nanos)</code> <code>void yield()</code>  <code>ClassLoader get/setContextClassLoader()</code> <code>String get/setName()</code> <code>int get/setPriority()</code> <code>ThreadGroup getThreadGroup()</code> <code>boolean isAlive()</code> <code>boolean is/setDataemon()</code> <code>boolean isInterrupted()</code>  <code>String toString()</code>
		<code>void checkAccess()</code> <code>void destroy()</code> <code>void interrupt()</code> <code>void join()</code> <code>void join(long millis)</code> <code>void join(long millis, int nanos)</code> <code>void run()</code> <code>void start()</code>

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

```
public class MiaClasseThread extends Thread{
    public void run(void){
        // metti qui il tuo codice
    }
}
```

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 scheduler deve eseguire quando il **thread** viene attivato.

### ESEMPIO *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 thrl = new ContaINanil();
        thrl.start();
        // o in una unica istruzione : new ContaINanil().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 thrl** creato dinamicamente dal precedente, con l'esecuzione dell'istruzione **thrl.start()** che lancia in concorrenza l'esecuzione del metodo **run()** del nuovo **thread**.

Una esecuzione produce sullo schermo il seguente output:

```
BlueJ: BlueJ: Terminale - TPSIT2 - UDA1 - □ ×
Opzioni
main
settenani
1 2 3 4 5 6 7
```

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 Due thread in esecuzione

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

```
public class ContaNani2 extends Thread{
    public ContaNani2(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 ContaNani2("topolino");
        Thread thr2 = new ContaNani2("pippo");
        thr1.start();
        thr2.start();
    }
}
```

Tre 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 **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 k = 0; k < volte; k++) {
            System.out.print ((k + 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 thrl = new Thread(cam1);
        thrl.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();
    }
}

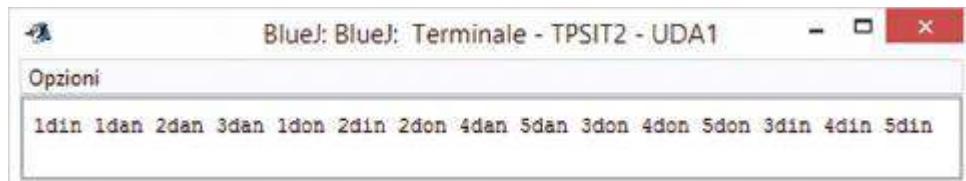
```

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:

```
1dan 2dan 3dan 4dan 5dan 1din 2din 3din 1don 4din 2don 5din 3don 4don 5don
```

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.  
Per esempio una sequenza possibile è la seguente:  
“din-dun” - don-dun - din-dun - dan-dun”

# ESERCITAZIONI DI LABORATORIO 11

## PRIORITÀ E PARAMETRI NEI THREAD JAVA

### ■ 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 in modo da far fare eseguire operazioni diverse in base a valori diversi. Non è possibile neppure utilizzare il metodo `start()`: anch'esso non ha parametri.

La **prima soluzione** che ci permette di passare parametri a un **thread** si ottiene utilizzando i parametri del metodo costruttore. Vediamo un esempio dove si vuole fa visualizzare ad un **thread** i numeri pari e un altro i numeri dispari fino al raggiungimento di un valore N.

Utilizziamo una variabile booleana (**pari**) che determinerà se il processo deve stampare i numeri pari o i numeri dispari. ►

```
public class PariDispari extends Thread{
    private int massimo;
    private boolean pari;
    private int ritardo = 500;

    public PariDispari (int finale, boolean pari){
        massimo = finale;
        this.pari = pari;
    }
}
```

Il metodo `run()` deve semplicemente controllare la variabile e comportarsi di conseguenza: per verificare il corretto funzionamento insieme al numero visualizziamo anche il nome del **thread** che ne effettua la stampa.

```
public void run(){
    String chisono;
    chisono = Thread.currentThread().getName();
    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(chisono+"-pari "+xx);
        }
        else{             // se è il thread che deve stampare i numeri dispari
            if (xx % 2 != 0) // numero dispari
                System.out.println(chisono+"-dispari "+xx);
        }
        try {Thread.sleep(ritardo);}
        catch (InterruptedException e){System.out.println(e);}
    }
}
```

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**.

Utilizziamo una variabile d'istanza per ogni dato che intendiamo “personalizzare” nei **thread** e passiamo tali parametri nel metodo costruttore, avendo letto il numero **n** dei numeri da visualizzare in input come argomento del **main()**:

```

1 public static void main(String[] args){
2     if (args.length != 1){
3         System.out.println("Dovresti passare il numero intero");
4         return;
5     }
6     int n = Integer.parseInt(args[0]);
7     Thread TP = new PariDispari (n + 1, true); // thread che conta i pari
8     Thread TD = new PariDispari (n + 1, false); // thread che conta i dispari
9     System.out.println(">Contate fino a " + n);
10    TP.start(); // avvio esecuzione thread
11    TD.start();
12    try{
13        TP.join(); // attesa terminazione thread
14        TD.join();
15    }
16    catch(Exception e){}
17    System.out.println("<-Fine conteggio!");
18 }
```

L'esecuzione produrrà un output simile al seguente: ►

Per ottenere l'alternanza corretta della visualizzazione dei numeri abbiamo dovuto inserire un ritardo: sappiamo che questo NON è un meccanismo di sincronizzazione bensì un “artificio” per far schedulare i **thread** alternativamente dal sistema operativo; vedremo come correggere questo esercizio nella prossima unità di apprendimento.

```

BlueJ Terminal Window - TPSIT2 - UDA1
Options
->Contate fino a 8
Thread-5-pari 0
Thread-6-dispari 1
Thread-5-pari 2
Thread-6-dispari 3
Thread-5-pari 4
Thread-6-dispari 5
Thread-5-pari 6
Thread-6-dispari 7
Thread-5-pari 8
<-Fine conteggio!

```

La **seconda soluzione** si ottiene mediante una **classe** nella quale si scrive il metodo costruttore con encapsulato la creazione del **thread** in modo da rendere possibile il passaggio di un parametro: definiamo una **classe Contatore()** con il costruttore che riceve due parametri, ne assegna i valori alla variabili locali e crea un **thread**.

```

1 class Contatore extends Thread{
2     private int massimo;
3     private boolean pari;
4     private Thread PID;
5     private int ritardo = 500;
6
7     public Contatore (int finale, boolean pari){
8         massimo = finale;
9         this.pari = pari;
10        PID = new Thread(this);
11        PID.start();
12    }
13 }
```

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 ;
        System.out.println(">Contate fino a " + n);
        Thread TP = new Contatore (n, true);
        Thread TD = new Contatore (n, false);
    }
}
```

L'output è praticamente 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** nella schedulazione della **CPU** secondo le politiche della **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: 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:

```

1 public class Priorita extends Thread{
2     private String chisono;
3
4     public Priorita (String nome){
5         setChieseI(nome);
6     }
7
8     public String getChieseI(){
9         return chisono;
10    }
11
12    public void setChieseI(String nome){
13        chisono = nome;
14    }
15
16    public void run(){
17        int conta = 0;
18        while(conta < 5000000){
19            conta++;
20            if((conta % 1000000) == 0)
21                System.out.println("Thread #"+chisono+", conta = "+conta);
22        }
23    }
24 }
```

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**):

```

1 public static void main (String [] args)
2 {
3     Thread TA = new Priorita ("IMPORTANTE (10)");
4     Thread TB = new Priorita ("poco importante");
5     TA.setPriority(Thread.MAX_PRIORITY);
6     TB.setPriority(Thread.MIN_PRIORITY);
7     TA.start();
8     TB.start();
9 }
```

Un'esecuzione genera il seguente output: ►

```

Thread #IMPORTANTE (10), conta = 1000000
Thread #IMPORTANTE (10), conta = 2000000
Thread #poco importante, conta = 1000000
Thread #IMPORTANTE (10), conta = 3000000
Thread #IMPORTANTE (10), conta = 4000000
Thread #poco importante, conta = 2000000
Thread #IMPORTANTE (10), conta = 5000000
Thread #poco importante, conta = 3000000
Thread #poco importante, conta = 4000000
Thread #poco importante, conta = 5000000

```

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!

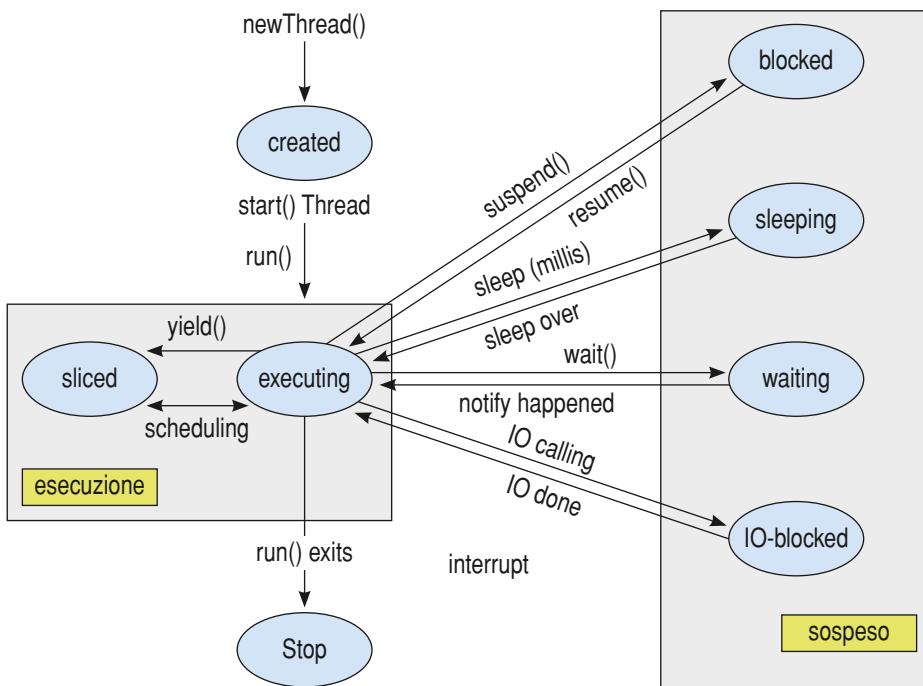
- 1 Realizza un programma in Java che attivi tre contatori indipendenti e ne visualizzi i valori (totali o parziali) in un intervallo massimo di 5 secondi.
- 2 Il contatore deve incrementare una proprietà interna (pubblica o privata) ed effettuare una visualizzazione a ogni ciclo.
- 3 Quindi a ogni contatore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

# ESERCITAZIONI DI LABORATORIO 12

## I THREAD JAVA: I METODI SLEEP, YIELD E JOIN

### ■ Metodi per i cambiamenti di stato

Nella lezione 3 abbiamo descritto i diversi stati che può assumere un **thread**: nella figura seguente riportiamo il diagramma completo degli stati di un **thread Java** indicando i nomi dei diversi metodi che possono causare 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**.

Vediamo alcuni esempi del loro utilizzo.

## Rilascio del processore

È consigliabile l'utilizzo di `stop()`, `suspend()` e `resume()` dato che non sono sicuri in quanto agiscono in modo diretto su un **thread** senza effettuare 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** avesse 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.

Il metodo `yield()`, invece, consente di trasferire il controllo del processore a un altro **thread** rimettendosi in coda, in attesa di poter riacquisire l'esecuzione: questa situazione prende il nome di **coroutining**.

## ■ Il metodo sleep()

Il metodo `sleep()` è un **metodo statico** che sospende il **thread** corrente e viene utilizzato per “far fare una pausa” al **thread** senza che questo utilizzi cicli del processore, cioè senza **attesa attiva**. La pausa di un **thread** può essere **interrotta** da un altro **thread** con il metodo `resume()`: in tale situazione viene sollevata un'eccezione **InterruptedException** e, quindi, è necessario che `sleep()` sia eseguito in un blocco **try-catch**.

Vediamo un semplice esempio dove introduciamo anche un meccanismo per riconoscere i **thread**.

### ESEMPIO

Indichiamo come costante il tempo di **attesa** che viene espresso in millisecondi e lo poniamo al valore di 200.

Ad ogni **thread** che viene avviato il sistema assegna un nome interno, che è costituito dalla parola **thread** seguita da un numero progressivo: questo individua univocamente i **thread** ed è possibile leggerlo mediante il metodo:

```
Thread.currentThread()
```

Viene restituito un dato di tipo **Thread** e lo memorizziamo in una variabile di classe **padre** che quindi conterrà l'identificatore del “padre di tutti i **thread**”.

```

1 public class EsempioSleep1 extends Thread{
2     static int tanti = 2;
3     static int attesa = 200;
4     private Thread padre;
5
6     public EsempioSleep1(){
7         padre = Thread.currentThread();
8     }
9
10    public void run(){
11        for (int x = 0; x < tanti; x++){
12            System.out.println("Nuovo thread");
13            printNome();
14            try {Thread.sleep(attesa);}
15            catch (InterruptedException e) {return;}
16        }
17        System.err.println("\n" + getName() + " finito");
18    }
19

```

Usiamo l'identificatore del **thread** per scrivere sullo schermo il “nome” del **thread** mediante il metodo **printNome()**: all'interno di questo confrontiamo con **this** il contenuto della variabile **padre** in modo da visualizzare sullo schermo se il codice viene eseguito dal **thread** padre oppure dal figlio.

```

1 public void printNome(){
2     Thread nome = Thread.currentThread();
3     if (nome == padre){
4         System.out.println("Thread padre");
5     }
6     else{
7         if (nome == this)
8             System.out.println("Nuovo thread");
9         else
10            System.out.println("Thread ignoto");
11    }
12

```

Facciamo ripetere un gruppo di istruzioni per un numero di volte indicato dalla variabile **tanti** sia nel **thread** che nel **main()**, e tra le istruzioni inseriamo l'istruzione che visualizza il nome del **thread** e la funzione **sleep(attesa)**:

- nel metodo **run()** all'interno di un il costrutto **try-catch**;
- nel **main()** rimandiamo la gestione dell'eccezione con la clausola **throws**.

```

1 public static void main(String[] args) throws InterruptedException{
2     new EsempioSleep1().start();
3     for(int x = 0; x < tanti; ++x){
4         System.out.println("Main thread");
5         Thread.sleep(attesa);
6     }
7 }

```

Solo un **thread** può essere in pausa e quindi non è possibile mettere in pausa un altro **thread** contemporaneamente al primo: se un secondo **thread** viene messo in **sleep()** automaticamente viene risvegliato quello che stava dormendo.

Una esecuzione del programma è la seguente:

```
Main thread
Nuovo thread
Nuovo thread
Main thread
Nuovo thread
Nuovo thread
Thread-1 finito
```

Dato che nel metodo `run()` abbiamo inserito come istruzione 17 di output il device `err` ci viene visualizzato il nome del **thread** nella finestra che riporta gli errori.

## ■ Il metodo `yield()`

Con il metodo `yield()` il **thread** non viene interrotto immediatamente: gli si segnala la necessità che esso si sospenda, lasciandogli però la scelta del momento opportuno, che sarà un momento non critico.

### ESEMPIO

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 thrl = new Thread(new Racchetta("ping"));
        thrl.start();
        // secondo giocatore
        Thread thr2 = new Thread(new Racchetta("pong"));
        thr2.start();
    }
}
```

L'eco sullo schermo è una sequenza infinita di scritte "ping pong" che si ripetono con ritardo di un secondo impostato dalla funzione `sleep()`.

L'osservazione dell'echo sullo schermo ci mostra però un comportamento indesiderato dato che non viene rispettata l'alternanza del ping con il pong: il problema è dovuto alla assenza di **sincronizzazione** e i singoli **thread** sono schedulati dal sistema operativo in modo "casuale". Nella prossima UA vedremo come regolare correttamente l'esecuzione dei **thread**.



## ■ 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.

### ESEMPIO

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();
    try{
        //attendo la loro terminazione
        thr1.join();
        thr2.join();
    }
    catch(InterruptedException e){
        System.out.println(e);
    }
}
```

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:

```

1 public class Aspettali implements Runnable{
2     String mionome;
3     int tempodormi;
4     Aspettali(String chisono, int quanto){
5         mionome = chisono;
6         tempodormi = quanto;
7     }
8
8     public void run(){
9         try{
10             System.out.println("il thread "+mionome+" ora si sospende ");
11             Thread.sleep(tempodormi);
12             System.out.println("il thread "+mionome+" si sveglia e termina ");
13         }
14         catch(InterruptedException e){
15             System.out.println(e);
16         }
17         return;
18     }
19 }
```

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 currentTimeMillis()**.

**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 currentTimeMillis()** fornisce proprio tale valore.

```

1 public static void main(String [] a){
2     long start = 0, stop = 0, delta = 0;
3     int quanti = 5000;
4     Thread thr1 = new Thread(new Aspettali("ali", quanti));
5     Thread thr2 = new Thread(new Aspettali("baba", quanti));
6     System.out.println("Ogni thread attende "+ quanti +" millisecondi");
7     //leggo il tempo alla creazione dei thread
8     start = System.currentTimeMillis();
9     //avvio l'esecuzione dei due thread
10    thr1.start();
11    thr2.start();
12    try{ //attendo la loro terminazione
13        thr1.join();
14        thr2.join();
15        //leggo il tempo alla fine dei thread
16        stop = System.currentTimeMillis();
17        System.out.println("il main riprende l'elaborazione ");
18    }
19    catch(InterruptedException e){
20        System.out.println(e);
21    }
22    delta = (stop - start) / 1000;
23    System.out.println("L'elaborazione è durata: "+delta+" secondi");
24 }
```

Il risultato dell'elaborazione è il seguente:

```
Ogni thread attende 5000 millisecondi
il thread baba ora si sospende
il thread ali ora si sospende
il thread ali si sveglia e termina
il thread baba si sveglia e termina
il main riprende l'elaborazione
L'elaborazione è durata: 5 secondi
```

Il tempo totale di attesa è di circa 5 secondi e non 10, che è la somma delle attese dei due processi: abbiamo così verificato che `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!

## ■ Un calcolo parallelo con `join()`

Scriviamo un programma che ci permette di valutare una espressione in modo parallelo. Supponiamo di dover risolvere:

$$k = 3 * (a-1) + 2 * (b-2) * 3 * (c-3)$$

e di far eseguire i singoli addendi a thread diversi:

```
thr1 → x = 3 * (a-1)
thr2 → y = 2 * (b-2)
thr3 → z = 3 * (c-3)
```

quindi:

$$k = (\text{thr1})\text{join}(\text{thr2})\text{join}(\text{thr3})$$

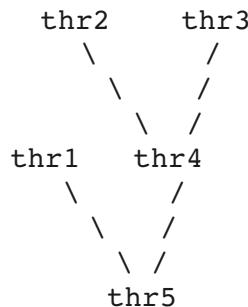
la somma finale viene ottenuta con due somme parziali, quindi nuovamente utilizzando due `thread` che aspettano i risultati parziali prima di poter operare:

```
thr4 → t = y * z
thr5 → k = x + t
```

cioè:

```
thr4 = (\text{thr2})\text{join}(\text{thr3})
thr5 = (\text{thr1})\text{join}(\text{thr4})
```

Il **grafo delle precedenze** si può rappresentare nel modo seguente



Scriviamo una **classe** apposita che contenga i dati iniziali **a**, **b**, **c** sui quali effettuare le elaborazioni e dove i singoli **thread** scrivano i risultati dei loro calcoli: ►

```

public class Buffer {
    public double x, y, z, t, k, a, b, c; // variabili condivise
    public Buffer() { // costruttore
        x = 0; y = 0; z = 0; t = 0; k = 0;
        a = 0; b = 0; c = 0;
    }
    public Buffer(double aa, double bb, double cc) {
        x = 0; y = 0; z = 0; t = 0; k = 0;
        a = aa;
        b = bb;
        c = cc;
        System.out.println(" I parametri valgono: a =" + a + " b =" + b + " c =" + c);
    }
}
  
```

Scriviamo una apposita **classe** per ciascun **thread** in modo che il costruttore legga dal buffer i dati che deve elaborare e il metodo **run()** effettui i calcoli e scriva il risultato sempre nel buffer. A titolo di esempio riportiamo quella che esegue la prima operazione parallela, cioè quella dalla quale istanziamo il **thr1**:

```

public class Operazionale extends Thread {
    /* Calcola x = 3 * (a - 1)
     *      x = c * (a - b) */
    Buffer dati;
    private double b = 1; // costante1
    private double c = 3; // costante2
    private double a; // parametro
    public Operazionale(Buffer d) {
        dati = d; // passaggio dell'oggetto con i dati
        a = dati.a; // valore di 'a'
    }
    public void run() {
        dati.x = c * (a - b);
        System.out.println(" Ho calcolato x : " + dati.x);
    }
}
  
```

Il programma principale si limita a creare i **thread**:

```

public class CalcoloParallelo {
    public static void main(String[] args) {
        double a, b, c;
        a = 2;
        b = 3;
        c = 4;
        System.out.println("Devo calcolare: 3*(a-1) + 2*(b-2) + 3*(c-3)");
        Buffer parziali = new Buffer(a, b, c); // init parametri
        Operazionale thr1 = new Operazionale(parziali); // singoli thread
        Operazionale thr2 = new Operazionale(parziali);
        Operazionale thr3 = new Operazionale(parziali);
        Operazionale thr4 = new Operazionale(parziali);
        Operazionale thr5 = new Operazionale(parziali);
    }
}
  
```

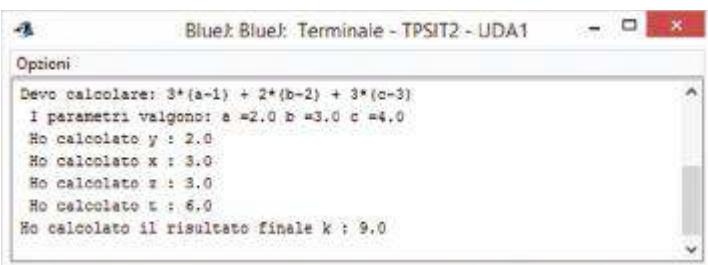
a mandarli in esecuzione e ad attenderne la loro terminazione:

```

17     thr1.start();           // calcolo concorrente
18     thr2.start();
19     thr3.start();
20     try { thr2.join(); }
21     catch(InterruptedException e) {System.out.println("Errore thr2");}
22     try { thr3.join(); }
23     catch(InterruptedException e) {System.out.println("Errore thr3");}
24
25     thr4.start();
26     try { thr1.join(); }
27     catch(InterruptedException e) {System.out.println("Errore thr1");}
28     try { thr4.join(); }
29     catch(InterruptedException e) {System.out.println("Errore thr4");}
30
31     thr5.start();
32     try { thr5.join(); }
33     catch(InterruptedException e) {System.out.println("Errore thr5");}
34
35 }
36 }
```

In **Java** è il programmatore che deve prestare attenzione a scrivere il codice in modo che venga rispettato l'ordine di avvio e di terminazione per la corretta realizzazione dei costrutti **fork-join** desiderati.

Una esecuzione del programma visualizza i seguenti risultati:



The terminal window displays the following output:

```

Bluel: Bluel: Terminale - TPSIT2 - UDA1
Opzioni:
Devo calcolare: 3*(a-1) + 2*(b-2) + 3*(c-3)
I parametri valgono: a =2.0 b =3.0 c =4.0
Ho calcolato y : 2.0
Ho calcolato x : 3.0
Ho calcolato z : 3.0
Ho calcolato t : 6.0
Ho calcolato il risultato finale k : 9.0

```



Thread e animazioni



**Prova adesso!**

- Costrutto fork-join

Dopo aver realizzato il diagramma delle precedenze, scrivi i programmi che implementando il costrutto **fork-join** eseguono le seguenti operazioni con il massimo grado di parallelismo leggendo come parametro i valori per i coefficienti **a**, **b** e **c**.

$$\begin{aligned}
 & 5 * [(2a + 4) * (7b + 3)] - 10c \\
 & (3 + 2a) * (5b - 7) + (8 - 3c) \\
 & (3 + a) - (5 - 2c) * (7by + 3) + 2a
 \end{aligned}$$

# 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 monoprocessoressi, 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**).

Ma anche in questo caso, cioè considerando solo macchine monoprocessoressi, 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 monoprocessoressi il **SO multitasking** permette l'esecuzione contemporanea di più **processi** che competono per accedere alle **risorse comuni**.

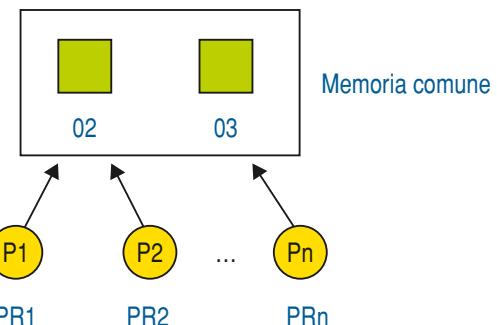
Possiamo individuare due modelli di **interazione concorrente** a prescindere dalla soluzione hardware, cioè sia che stiamo analizzando una architettura distribuita oppure una situazione di multitask su macchina **SISM**:

- ▶ 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 monoprocessoressi 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

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.

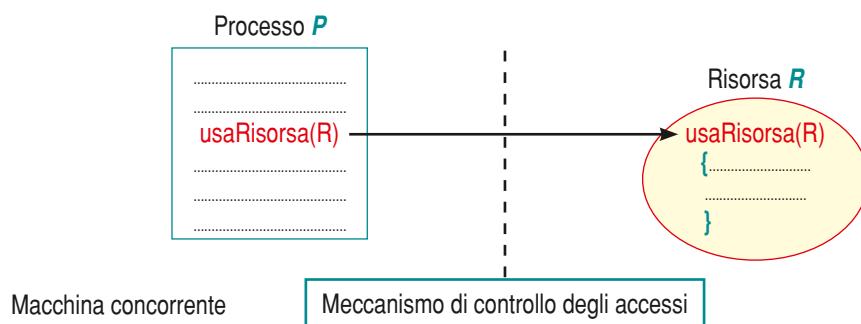
<b>Risorse dedicate</b> (visibili in ogni istante da un solo processo)		<b>Risorse condivise</b> (visibili in ogni istante anche da più processi contemporaneamente)
risorse allocate staticamente	risorse private	risorse comuni
risorse allocate dinamicamente	risorse comuni	risorse comuni

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 queste 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**.

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**.





### MUTUA ESCLUSIONE

L'accesso a una **risorsa** si dice **mutuamente esclusivo** se a ogni istante, al massimo un **processo** può accedere a quella **risorsa**.

Vediamo nelle diverse situazioni come interagiscono i processi nella situazione di **condivisione delle risorse**, cioè quando **competono**, quando **cooperano** oppure quando **interferiscono** nelle due situazioni di allocazione **statica** e **dinamica**.

## Competizione

La **competizione** è una interazione tra **processi prevedibile** e **NON desiderata**.

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**.

La soluzione della **competizione** avviene mediante la **sincronizzazione indiretta o implicita**, gestita dal **sistema operativo**.

## Cooperazione

La **cooperazione** è una interazione tra **processi prevedibile** e **desiderata** dato che è insita nella logica del programma.

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) scrive un dato nella risorsa (**produttori**) e un altro **processo** (o più processi) legge 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 dato "prodotto" dal primo **processo** prima che venga "consumato" dal secondo **processo**.

La soluzione della **cooperazione** avviene mediante **sincronizzazione diretta o esplicita**, gestita dal **sistema del programmatore** mediante **scambio di informazioni**.

## Interferenza

L'interferenza è dovuta alla **competizione** che avviene tra **processi** che utilizzano senza le opportune autorizzazioni **risorse condivise** oppure a una erronea soluzione dei problemi di **competizione** e di **cooperazione**.

L'interferenza è una interazione tra **processi NON prevedibile** e **NON desiderata**.

Per esempio possiamo essere in una delle due seguenti situazioni:

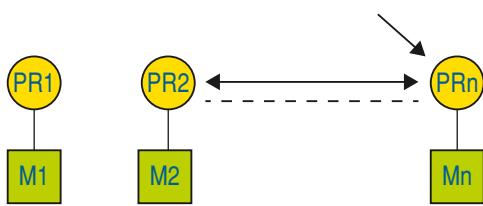
- P1 chiama P2 senza necessità e lo rallenta;
- P1 chiama P2 ma al momento sbagliato.

Generalmente i malfunzionamenti sono legati alla velocità relativa dei **processi** e quindi i risultati sono "dipendenti dal tempo": di conseguenza è molto complesso effettuare il debug.

## ■ 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:

- Ⓐ utilizzare linguaggi che prevedono costrutti esplicativi per realizzare lo scambio di messaggi, come il **CSP** (**Communicating Sequential Processes**) proposto da ▲ **Tony Hoare** ▷;
- Ⓑ 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**:

- Ⓐ 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” ▷:

D **stretto**: se si limita alla trasmissione di un messaggio dal mittente al destinatario;

D **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:

D comunicazione di tipo **simmetrico** e sincrono a **rendez-vous stretto** tipico del **CSP**;

D 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. 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 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.

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 servitorea 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).

**AREA digitale**



Classificazione di Flynn

# 2

# Comunicazione e sincronizzazione

- L1 **La comunicazione tra processi**
- L2 **La sincronizzazione tra processi**
- L3 **I semafori**
- L4 **Applicazione dei semafori**
- L5 **Il problema dei produttori/consumatori**
- L6 **Il problema dei lettori/scrittori**
- L7 **Il problema del deadlock: banchiere e filosofi a cena**
- L8 **I monitor**
- L9 **Lo scambio di messaggi**

## Esercitazioni di laboratorio

- 1 La comunicazione con segnali asincroni; 2 Thread e schedulazione
- 3 I semafori binari in C; 4 La soluzione del deadlock in C; 5 La soluzione del problema produttori/consumatori con i semafori; 6 Variabili condizione; 7 I monitor con le variabili condition in C; 8 I monitor con i semafori in C; 9 I semafori in Java; 10 I monitor in Java; 11 Un esempio con i Java thread; 12 I deadlock in Java

### Conoscenze

- Conoscere il modello ad ambiente globale e locale
- Comprendere l'esigenza di sincronizzazione
- 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 di starvation e di deadlock
- Comprendere le proprietà di safety, di fairness e di liveness

### Competenze

- Individuare le tipologie di errori nei processi paralleli
- Definire e utilizzare i semafori di basso livello e spin lock()
- Utilizzare gli strumenti di sincronizzazione per thread in C
- Utilizzare le condition variable in C
- Implementare i monitor in C
- Utilizzare gli strumenti di sincronizzazione per thread in C
- Implementare i monitor in Java

### Abilità

- Risolvere le situazioni di starvation
- Risolvere le situazioni di deadlock
- Risolvere i problemi produttore/consumatore in C
- Risolvere il problema dei filosofi in C
- Risolvere i problemi produttore/consumatore in Java
- Risolvere il problema dei filosofi con il linguaggio in Java

## AREA *digitale*

-  Esercizi
-  Classificazione di Flynn
- ▶ Applicazioni in real time
- ▶ Esempio riepilogativo (non informatico)
- ▶ Grafo di Holt e grafo di attesa
- ▶ Tabella dei segnali UNIX/LINUX
- ▶ Quando viene notificato/gestito un segnale
- ▶ Gruppi di processi
- ▶ Impostare una sveglia
- ▶ Nota per gli utenti MAC
- ▶ Soluzione del problema del produttore/consumatore con i mutex
- ▶ Mutex per la gestione di vincoli di precedenza
- ▶ Esercizi per il recupero
- ▶ Esercizi per l'approfondimento

 Soluzioni (prova adesso, esercizi, verifiche)

Puoi scaricare il file anche da  [hoepliscuola.it](http://hoepliscuola.it)

## Verifichiamo le conoscenze



### 1. Risposta multipla

**1 Il parallelismo dell'esecuzione di processi concorrenti:**

- a) è solo virtuale
- b) è solo reale
- c) è reale su sistemi multiprocessor e virtuale su sistemi multiprocessor
- d) è reale su sistemi multiprocessor e virtuale su sistemi multiprocessor
- e) nessuna delle precedenti

**2 Nella classificazione di Flynn il personal computer è**

- |                      |                      |
|----------------------|----------------------|
| a) una macchina SISD | c) una macchina MISD |
| b) una macchina SIMD | 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      | c) sempre comuni       |
| b) private se dedicate | d) comuni se condivise |

**6 Le risorse allocate dinamicamente sono:**

- |                        |                        |
|------------------------|------------------------|
| a) sempre private      | c) sempre comuni       |
| b) private se dedicate | d) comuni se condivise |

**7 Nel modello a scambio di messaggi le risorse comuni:**

- |                                |  |
|--------------------------------|--|
| a) sono allocate staticamente  | c) non ci sono risorse condivise                         |
| b) sono allocate dinamicamente | d) sono allocate in maniera dipendente dall'architettura |

**8 Quali tra i seguenti sono tipi di comunicazione nei modelli a scambio di messaggi?**

- |                                   |                                   |
|-----------------------------------|-----------------------------------|
| a) sincrono a rendez-vous stretto | c) asincrono a rendez-vous esteso |
| b) sincrono a rendez-vous stretto | d) asincrono a rendez-vous esteso |



### 2. Vero o 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 esplicativi per realizzare lo scambio di messaggi. **V F**
- 11** Nella comunicazione simmetrica il mittente nomina esplicitamente il destinatario e viceversa. **V F**

### 3. Completamento

multitask • interazione concorrente • scambio di messaggi • SISM • memoria comune • architettura distribuita • interazione • processi • risorse • sincronizzazione • risorsa condivisa • non divisibile • sezioni critiche • esteso • allocate staticamente • mutualmente esclusivo • stretto • sincronizzino • condivise • condivise • mutua esclusione • senza • allocate staticamente • rendez-vous • sincrona • asincrona

- 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.

# 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:

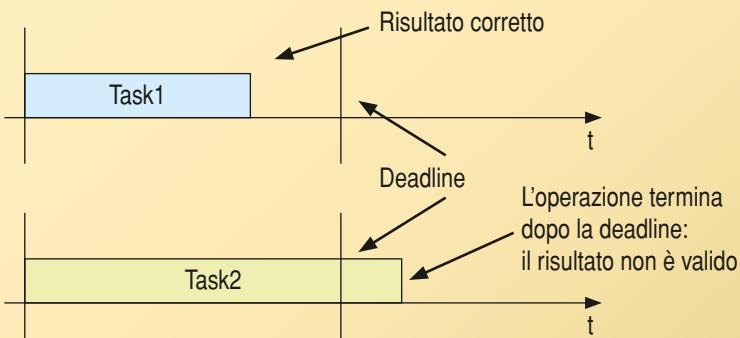
- ▶ il **programma sequenziale** produce diversi risultati ma solo in funzione di dati di input diversi e il programmatore, mediante casi di prova, può verificare i comportamenti del programma confrontando gli output con i risultati attesi: non è possibile in questo modo dimostrare la correttezza totale 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":

- A determinismo:** i risultati devono essere uguali per ogni esecuzione, indipendentemente dalla sequenza di schedulazione;
- B timing constraint:** i risultati devono essere prodotti entro certi limiti temporali fissati (**deadlines**) (specificata 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.

**AREA digitale**

Applicazioni in real time

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.

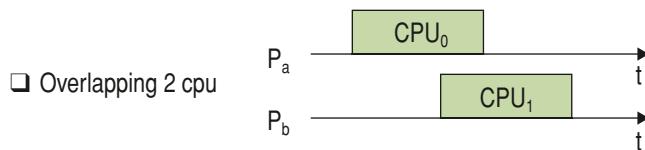
## ■ Definizioni e proprietà

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.

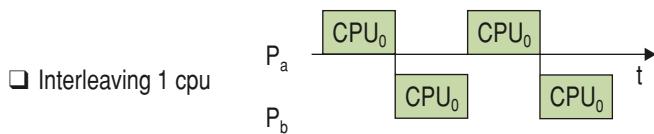
### Interleaving e overlapping

Innanzitutto 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 monoprocessoressi.

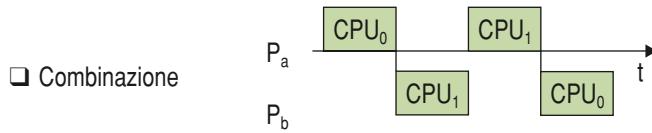
- 1 Nei **sistemi multiprocessoressi** più **processi** vengono eseguiti simultaneamente su **processori** diversi e quindi sono “*sovraposti nel tempo*”: in questo caso si definisce **overlapping** la **sovraposizione temporale** di **processi**.



- 2 Nella macchina con un **singolo processore** i processi sono “*alternati nel tempo*” ma con velocità tali da dare l'impressione di avere un multiprocessoresso: 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).



- 3 Potrebbe anche esserci una situazione in cui sono presenti **più macchine**, ciascuna **multitasking** e quindi avere una **combinazione delle due situazioni** sopra descritte di **interleaving/overlapping**.



Il caso 1 implica che ogni **azione** sia svolta da un distinto esecutore fisico (**processore**) ed equivale a un **PARALLELISMO REALE**:

$p$  **processori** per  $a=p$  azioni

Negli altri due casi siamo in situazioni in cui si dispone di un **numero  $p$**  di **processori** inferiore al **numero  $a$**  di **azioni** da eseguire ed equivale a un quasi-parallelismo o a un **PARALLELISMO VIRTUALE**:

$p$  **processori** per  $a>p$  azioni

## 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  $A, B, \dots, X, Y, \dots$  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 **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**:

- ▷  $\text{range}(i_a) \cap \text{range}(i_b) = \emptyset$
- ▷  $\text{range}(i_a) \cap \text{domain}(i_b) = \emptyset$
- ▷  $\text{domain}(i_a) \cap \text{range}(i_b) = \emptyset$

Non si impone alcuna condizione sulla intersezione dei domini delle due istruzioni.

### ESEMPIO

### Verifica delle condizioni di Bernstein

Vediamo tre semplici esempi dove analizziamo due istruzioni alla volta:

1	Istruzione	Domain(istruzione)	Range(istruzione)
A	$X \leftarrow Y + 5;$	Y	X
B	$X \leftarrow Y - 3;$	Y	X

violano la condizione 1:

- ▷  $\text{range}(A) \cap \text{range}(B) = X$  che è diverso dall'insieme vuoto  $\emptyset$

2	Istruzione	Domain(istruzione)	Range(istruzione)
A	$X \leftarrow Y + 2;$	Y	X
B	$Y \leftarrow X - 1;$	X	Y

violano la condizione 2 e la condizione 3:

- ▷  $\text{range}(i_a) \cap \text{domain}(i_b) = X$  che è diverso dall'insieme vuoto  $\emptyset$
- ▷  $\text{domain}(i_a) \cap \text{range}(i_b) = Y$  che è diverso dall'insieme vuoto  $\emptyset$

3	Istruzione	Domain(istruzione)	Range(istruzione)
A	scrivi (X)	X	$\emptyset$
B	$X \leftarrow X + Y + 3;$	X, Y	X

violano la condizione 3:

- ▷  $\text{domain}(i_a) \cap \text{range}(i_b) = X$  che è diverso dall'insieme vuoto  $\emptyset$

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 *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) <b>ESECUZIONE SEQUENZIALE</b>	(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**.

La regola di **mutua esclusione** impone che le operazioni con le quali i **processi** accedono alle variabili comuni **non si sovrappongano nel tempo**: **inoltre nessun vincolo è imposto sull'ordine** con il quale le operazioni sulle variabili vengono eseguite.



### 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**.

Nei nostri programmi le **risorse comuni** condivise saranno le **variabili globali** che verranno utilizzate dai diversi processi per scambiarsi le informazioni (**modello a memoria condivisa**).

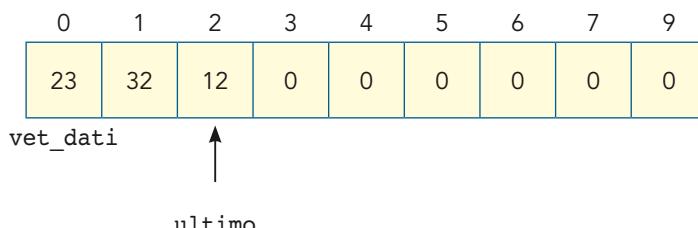
Vediamo un esempio nel quale due processi si scambiano dati attraverso la **memoria condivisa** per comprendere meglio perché è di fondamentale importanza regolare l'accesso a tale risorsa in modo **esclusivo**.

#### ESEMPIO

#### *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
<pre>funzione inserisci(nuovo_dato) inizio     ultimo ← ultimo+1     vet_dati[ultimo] ← nuovo_dato fine</pre>	<pre>funzione preleva() inizio     letto r vet_dati[ultimo]     vet_dati[ultimo]≤ 0     ultimo r ultimo -1 fine</pre>

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.

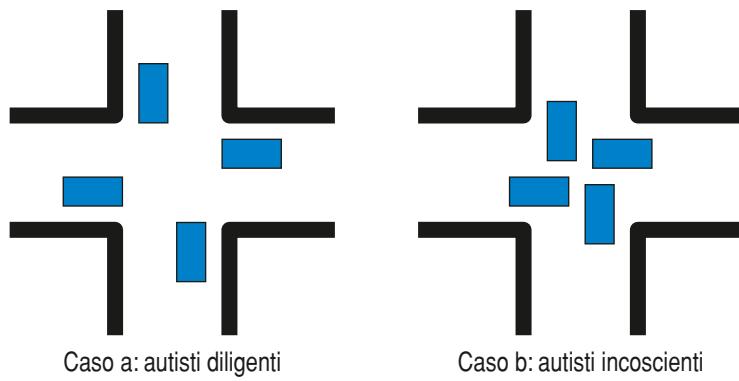
Vediamo un terzo esempio dove la mutua esclusione permette di risolvere il problema della interferenza ma può causare il blocco permanente dei processi.

### ESEMPIO **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.



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 <b>indiretta</b> l’uno dell’altro	Cooperazione (sharing)	Sincronizzazione	Mutua esclusione Deadlock Starvation
Processi con conoscenza <b>diretta</b> l’uno dell’altro	Cooperazione (comunicazione)	Comunicazione	Deadlock Starvation

Esistono molte soluzioni a questo problema, a seconda della **politica** di accesso alla **risorsa** che viene scelta, che può favorire nell’accesso una classe di **processi** oppure un’altra.

## ■ Proprietà non funzionali: safety e liveness

Nella soluzione dei problemi con processi interagenti non è quindi sufficiente programmare in modo da soddisfare solamente gli **aspetti funzionali** per affermare che un programma è **corretto**: per esempio, non basta verificare che i calcoli diano risultati esatti è che l'algoritmo sia giusto per essere sicuri che sempre “tutto funzioni”.

È necessario tener presente anche degli aspetti **NON funzionali** che garantiscono la correttezza in ogni situazione di schedulazione e di interazione, cioè per ogni possibile “storia di esecuzione” del programma stesso: tra queste ricordiamo il rispetto del tempo di esecuzione di un’applicazione in un sistema real-time, la sicurezza di un’applicazione, ma anche l’assenza di **deadlock** e di **starvation**.

**Premessa:** il **sistema operativo** deve garantire la proprietà di **fairness**, cioè deve sempre mandare in esecuzione qualsiasi **processo** soddisfacendo prima o poi tutte le richieste di esecuzione così da non essere lui la causa di eventuali situazioni di **starvation**.

Le **proprietà non funzionali** fondamentali che un **programma concorrente** deve soddisfare sono due che descriviamo qui di seguito: la prima riguarda le **risorse (safety)** mentre la seconda riguarda i **processi (liveness)**.

### Safety

La **safety** è una proprietà che riguarda lo **stato delle risorse**: 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, questo va “nella direzione voluta” senza eseguire azioni indesiderate, cioè “**nothing bad ever happens**”!

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** in modo che tutte le risorse del sistema siano sempre in uno **stato consistente**: si devono eliminare le **interferenze indesiderate** tra i **processi**.

Tra le molte possibili cause di errore e di interferenza tra processi che possono violare la **safety** ricordiamo:

- la presenza di corse critiche causate da una non corretta **mutua esclusione**;
- l’accesso e le azioni su stati di **risorse non consistenti**;
- la violazioni dell’**atomicità** di certe operazioni (**lock** e **P()** su semafori);
- esecuzione di azioni che dovrebbe essere proibite ma permette dal linguaggio di programmazione utilizzato per scrivere i programmi;
- esecuzione di operazioni su valori e dati non aggiornati, mantenuti in cache e quindi non consistenti.

Molti linguaggi di programmazione sequenziale sono **type-safe** perché il compilatore controlla il corretto uso dei tipi di dato, ma questo controllo viene fatto staticamente, a **compiled-time**: i sistemi **multi-threaded** introducono la **dimensione del tempo** e quindi non è sufficiente il controllo statico ma si devono quindi utilizzare tecniche di programmazione che preservino la consistenza degli oggetti evitando interferenze a **run-time**.

## Liveness

La **liveness** è una proprietà che riguarda le attività, cioè i **processi**, in merito alla loro “esecuzione ed evoluzione”: si deve garantire che il processo che avanza porterà a termine in modo corretto il proprio lavoro, cioè “**something good eventually happens**”. A tal fine 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 “**bloccino**” in attesa di eventi che non possono verificarsi: è la proprietà che esclude situazioni di **deadlock**.

La **liveness** garantisce che prima o poi **tutti i processi** entrano in uno stato corretto e progradiscono verso il completamento, cioè si deve assicurare la corretta **cooperazione** tra i **processi** evitando situazioni di **deadlock** e di **starvation**.

Tra le molte possibili cause di errore e **interferenza** tra **processi** che possono violare la **liveness** di una applicazione possono essere:

- ▶ errori di **sincronizzazione**;
- ▶ errori con i segnali tra **processi cooperanti**;
- ▶ fallimenti di un **processo**: si attende un segnale da un altro **processo** che è andato in crash;
- ▶ **livelock**: continuo tentativo di una azione che fallirà sempre;
- ▶ **lockout**: chiamata di una operazione che non sarà mai disponibile;
- ▶ esaurimento di una risorsa necessaria per evolvere.

Ciascuna di queste situazioni può portare o alla **starvation** di un **processo** oppure addirittura al **deadlock** di tutto il sistema.

Ricapitolando, in caso di violazione delle proprietà sopra descritte si generano problemi di:

- ▶ **violazione fairness**: si ha l’assegnazione di risorse in modo non equo a tutti processi;
- ▶ **violazione liveness**:
  - **starvation**: una classe di processi non entra mai in possesso della risorsa;
  - **deadlock**: blocco di tutti i processi che si ostacolano a vicenda.

## ■ 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**. Si noti inoltre che **safety** e **liveness** sono proprietà “in contrasto” tra loro: la realizzazione di una corretta applicazione concorrente deve bilanciare le diverse esigenze di progettazione e introdurre sincronizzazioni troppo forti che possono ridurre la concorrenza di esecuzione impattando sulle prestazioni (per esempio, l’uso di **attese attive** su eventi che devono accadere o in attesa di segnali da parte di altri processi comporta spreco di risorse, soprattutto di tempo CPU).

Nelle prossime lezioni descriveremo gli strumenti e le tecniche per scrivere programmi paralleli e garantire l’assenza di situazioni di **starvation** o di **deadlock**.

**AREA digitale**



Esempio riepilogativo  
(non informatico)

# 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 sia del primo caso, cioè di situazioni in cui la **competizione** tra i processi ci porta a **interferenze** che possono provocare situazioni inaccettabili e che quindi devono

essere gestite con la **mutua esclusione**, sia del caso in cui i processi collaborano e quindi devono scambiarsi informazioni attraverso aree di memoria condivisa e con accesso regolato da meccanismi di **sincronizzazione**.

Il **frammento di programma** che utilizza la **risorsa R** che deve essere gestita in **mutua esclusione** si dice **sezione critica** o **regione critica** rispetto alla **risorsa R**: è necessario garantire che un **processo** acceda alla risorsa da solo, cioè che quindi esegua la **sezione critica** con la certezza di essere l'unico utilizzatore che volta per volta esegue il codice che utilizza la **risorsa**.

La **sezione critica** viene gestita in modo che:

- Ⓐ 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;
- Ⓑ 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:

- Ⓐ la **risorsa** o è libera oppure è utilizzata da un solo processo (condizione di **mutua esclusione**);
- Ⓑ i **processi** devono sempre poter accedere alla **risorsa** richiesta e portare a termine il proprio lavoro (**condizione di fairness**);
- Ⓒ 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.

I meccanismi che permettono di regolare l'accesso alla **regione critica** risolvendo di fatto il problema della **mutua esclusione** sono essenzialmente tre:

- ▷ gli **spin lock** (o **semafori binari**);
- ▷ **P(S) & V(S)** ovvero i **semafori di Dijkstra**;
- ▷ i **monitor**.

In questa lezione affronteremo il 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.

## ■ Semafori di basso livello e spin lock()

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 **spin lock**.

Vediamo come esprimere in pseudocodifica i segmenti di codice che effettuano 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)
inizie
    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 *Il primo semaforo*

Dichiarazione di un oggetto di tipo **semaphore**:

```
semaphore s = vi;
```

dove vi ( $vi \geq 0$ ) 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 *Semafori ferroviari*

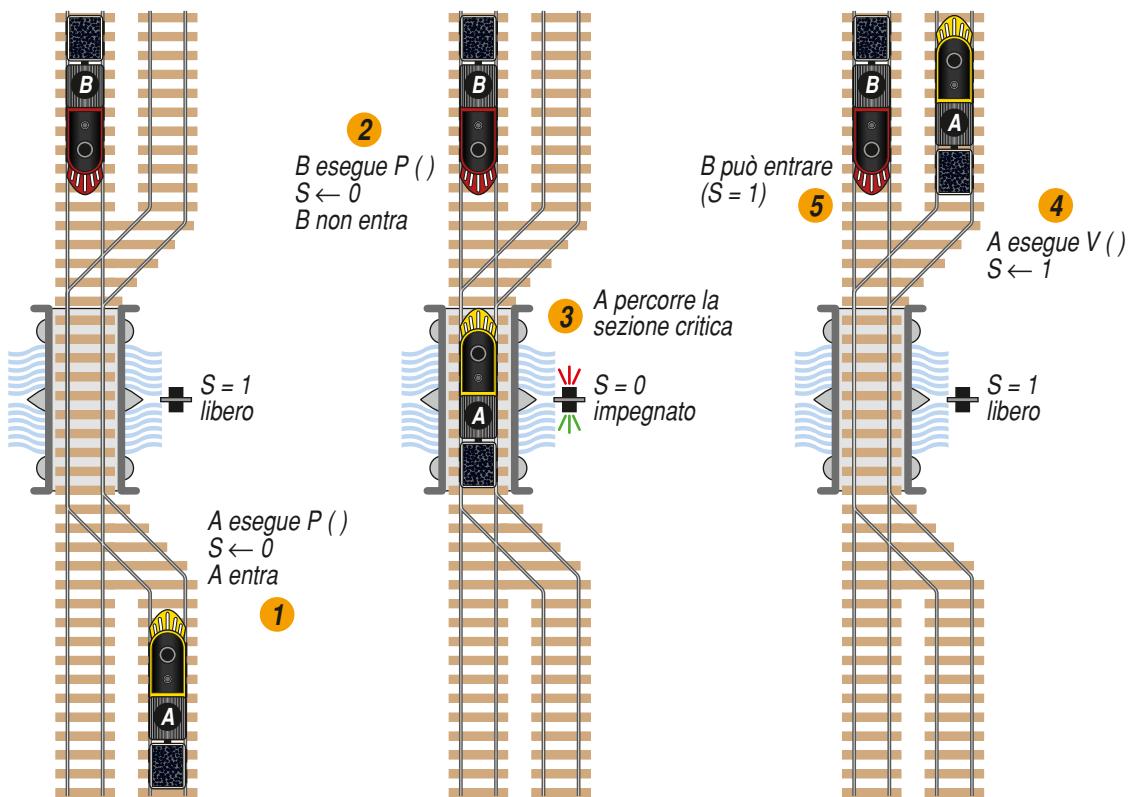
Il nome “**semaforo**” proviene dalla segnalazione ferroviaria e facciamo un esempio di gestione rimanendo in ambito ferroviario: supponiamo di avere la situazione rappresentata in figura, dove una linea a due binari a un certo punto deve confluire su di un ponte con binario unico.

Il ponte è la **risorsa** condivisa che dovrà essere gestita in **mutua esclusione**.

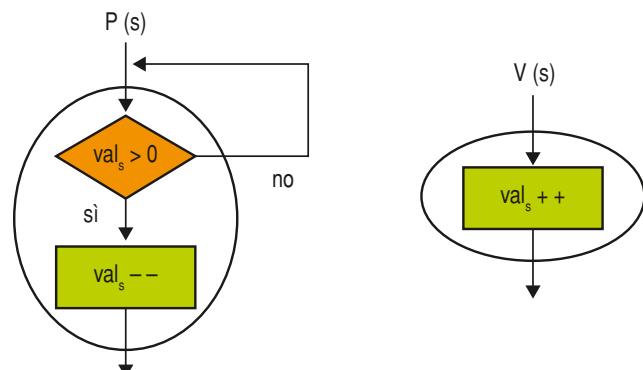
Associamo al ponte un semaforo che viene controllato e gestito dai macchinisti dei treni: quando un treno A si avvicina al ponte osserva il semaforo:

- 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 quello a fianco: ►



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)
inizialia
    se <è presente un processo in attesa>
        allora
            <poni il primo processo nello stato di pronto>
            S ← S+1;           // rilascia la risorsa
fine
```

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)
inizialia
    <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)
inizialia
    <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.



## 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.

## ■ 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  $x \leq \text{NUM}$  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**.

# Applicazione dei semafori

In questa lezione impareremo...

- ▶ a realizzare la mutua esclusione mediante i semafori
- ▶ a regolare l'accesso 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

inizial /* 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 *Prenotazione posti al cinema*

In una sala cinematografica con più casse per la vendita dei biglietti, il numero di posti è continuamente aggiornato e memorizzato in una variabile intera condivisa **postiLiberi** (inizializzata a 200). Lo spettatore richiede **quanti** posti ( $> 0$ ): se è possibile soddisfare la richiesta, i posti vengono occupati aggiornando **postiLiberi** 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** (per esempio la lettura di dati) ma affinché un **processo** possa eseguire una operazione diversa da quella che stanno eseguendo tutti gli altri è necessario che tutti questi finiscano di utilizzare la **risorsa condivisa** e che questa quindi risulti “libera da ogni processo”.

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
    P(mutexPDK)
        <controllo e aggiornamento ProcessiDentroK >           // prologo
    V(mutexPDK)
        <corpo della funzione operazioneK >
    P(mutexPDK)
        <controllo e aggiornamento ProcessiDentroK >           // parte centrale
    V(mutexPDK)
        <controllo e aggiornamento ProcessiDentroK >           // epilogo
fine
```

Le operazioni effettuate dal prologo sono di seguito elencate e commentate:

```
P(mutexPDK)                      //accesso in mutua esclusione alla RC
    ProcessiDentroK++;
    if (ProcessiDentroK ==1)
        P(mutex)
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 --
    if (ProcessiDentroK ==0)
        V(mutex)
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
```

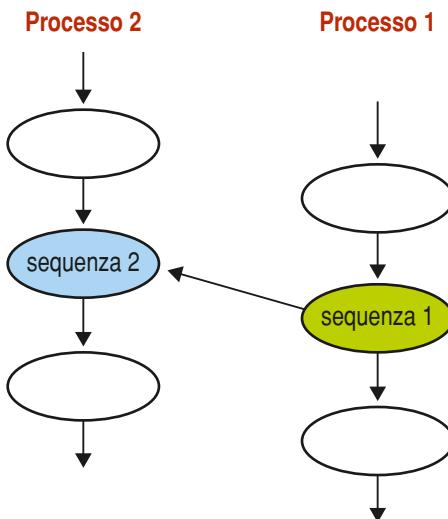
È immediata l'estensione nel caso di un insieme di attività simili da eseguirsi contemporaneamente: basta scrivere una procedura per ogni operazione e aggiungere un contatore e un semaforo riservato per regolarne l'accesso a ciascuna di esse.

Deve invece rimanere unico il semaforo **mutex** che regola l'accesso alla risorsa comune elaborata all'interno del **<corpo della funzione operazioneK>**.

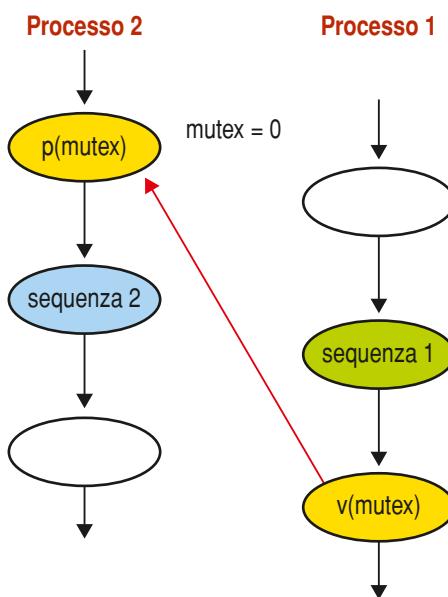
## ■ Semafori come vincoli di precedenza

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**.

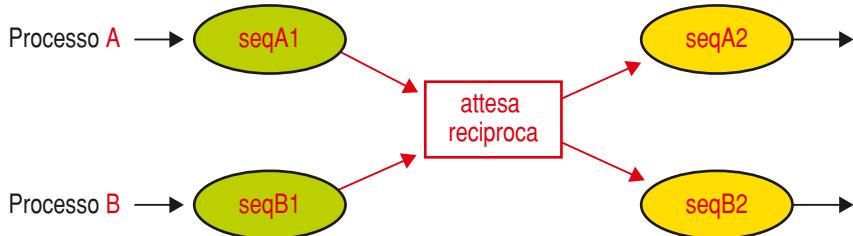


Introduciamo un semaforo **mutex** che inizializziamo a rosso (**mutex=0**) e che viene messo a verde dal **processo1** solo dopo che ha eseguito la **sequenza1**: il secondo processo, prima di eseguire la **sequenza2**, effettua un test su questo semaforo che troverà *verde* solo dopo l'esecuzione della **sequenza2** altrimenti, trovandolo *rosso*, aspetta il **processo1**.



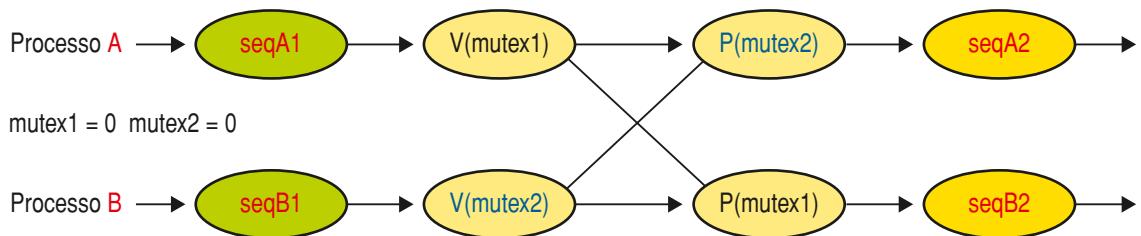
## ■ Problema del rendez-vous

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 sequenze **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 successivamente 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**.

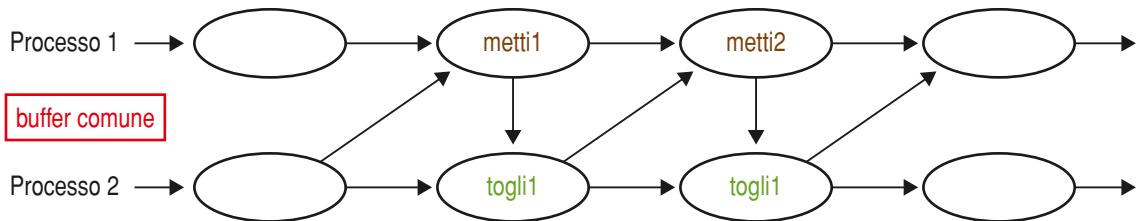
### ESEMPIO Rendez-vous prolungato

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
    
```

# 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 nelle quali due (o più) **processi** si ostacolano a vicenda impedendo reciprocamente di portare a termine il proprio lavoro: il **deadlock** viene anche chiamato “abbraccio mortale” in quanto l'**interferenza** porta al fallimento di entrambi e... “alla loro morte”.



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 nessun altro processo che le necessita per poter evolvere.

Non tutti i problemi di **concorrenza** possono portare a situazioni di **deadlock**: affinché ci sia un **deadlock** nel sistema devono verificarsi contemporaneamente **quattro condizioni** di seguito descritte, che sono state dimostrate da **Coffman** nel 1971 come le condizioni **necessarie e sufficienti** affinché esista la possibilità di verificarsi una situazione di **blocco critico** o **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 solo quando hanno terminato di usarle;
- 3 richieste bloccanti** (detta anche “◀ **hold and wait** ▶”): le richieste devono essere bloccanti, e un processo che ha già ottenuto alcune **risorse** può chiederne ancora altre;
- 4 attesa circolare**: devono essere presenti nel sistema almeno due processi e ciascuno di essi è in attesa di una risorsa occupata dall'altro.

◀ **Hold and wait** A process or thread holding a resource while waiting to get hold of another resource. ►

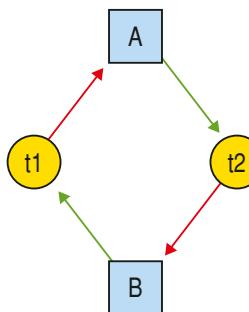


In questa lezione analizzeremo le tecniche che possono essere utilizzate per gestire il problema del **deadlock**.

## ■ 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 seguente:



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 il **thread** **t1** richiede una **risorsa** A che è occupata da **t2** e il **thread** **t2** richiede una **risorsa** B che è occupata da **t1**!

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.

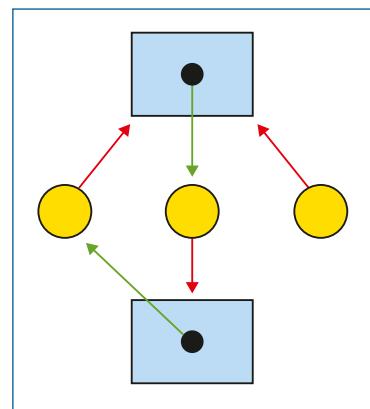
Nel caso in cui le risorse sono presenti con molteplicità superiore a 1, la presenza di un ciclo nel caso di Holt non è condizione sufficiente per avere deadlock.

### AREA digitale

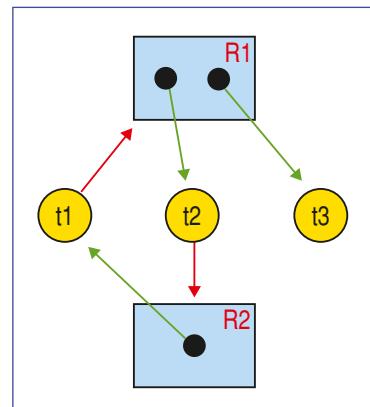


Grafo di Holt e grafo di attesa

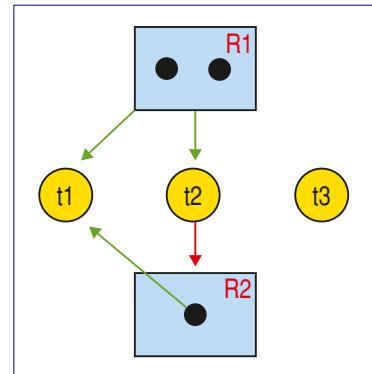
Lo possiamo verificare con i seguenti esempi:



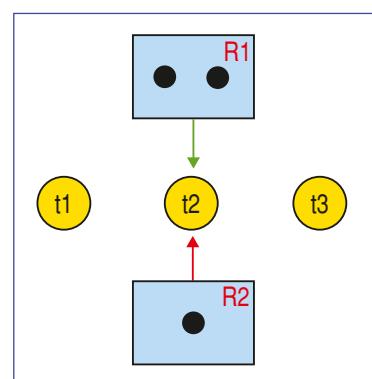
In questa situazione la presenza di **un ciclo** indica la *presenza di un deadlock*. ►



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**: ▶

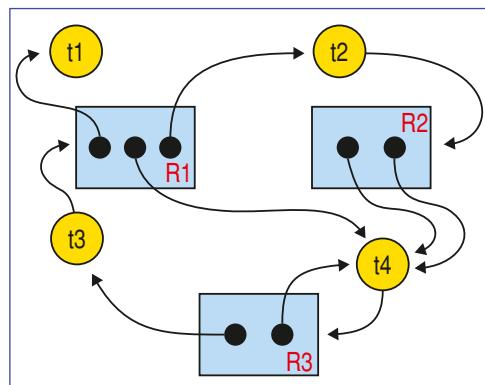


Questo potrà evolvere e rilasciare entrambe le risorse al **thread t2** che può portare a termine le sue elaborazioni. ▶

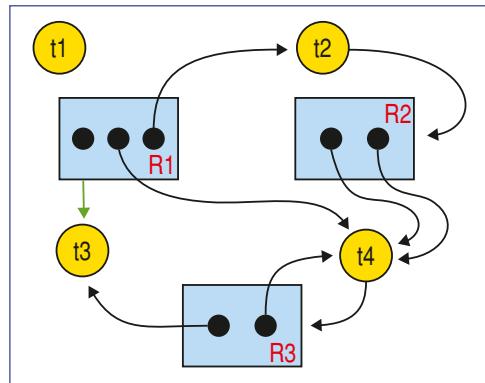


### ESEMPIO Riduciamo un grafo-complesso

Analizziamo un esempio più articolato per scoprire se è possibile che si verifichi un **deadlock**. ▶



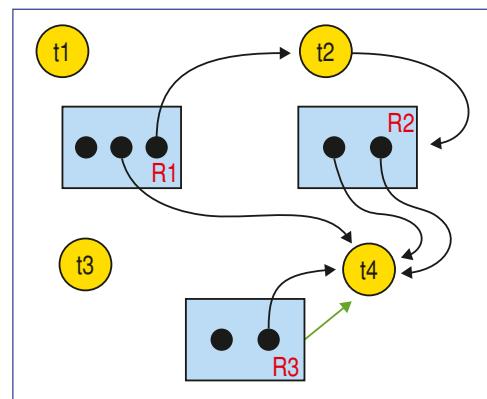
Per prima cosa osserviamo che il **thread t1** ha assegnato l'unica risorsa che richiede e quindi è in grado di portare a termine il proprio lavoro, quindi lo riduciamo: ▶



Dopo aver effettuato la riduzione di **t1**, possiamo riassegnare la **R1** da lui posseduta a **t3**.

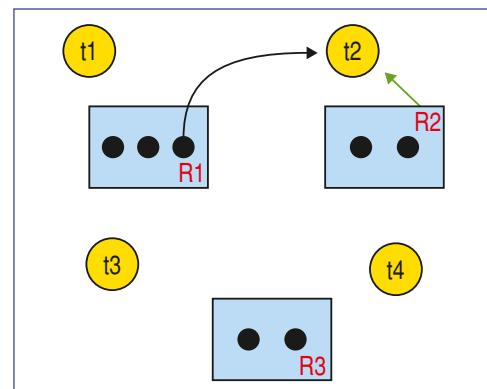
Ora è il **thread t3** che può evolvere e quindi lo riduciamo: ▶

Dopo aver effettuato la riduzione di **t3**, possiamo riassegnare la **R3** da lui posseduta a **t4**.

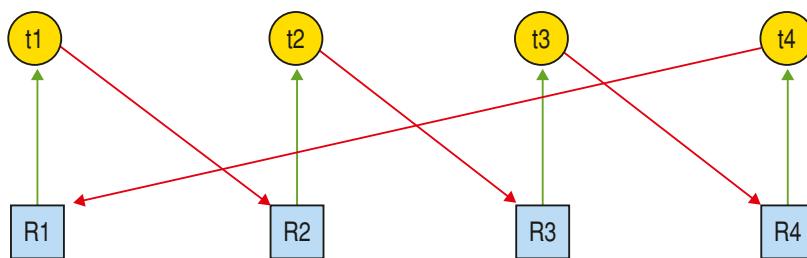


A questo punto anche il **t4** ha a disposizione tutte le risorse che gli necessitano, e lo possiamo ridurre: ▶

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 riportato di seguito, dove esiste una situazione che coinvolge ciclicamente 4 **thread**.



## ■ Come affrontare lo stallo

Lo **stallo** può essere affrontato sostanzialmente in quattro diverse strategie:

- 1 detection e recovery:** riconoscerlo ed eliminarlo;
- 2 avoidance:** evitarlo con specifiche politiche di allocazione;
- 3 prevention:** impedirlo rimuovendo una delle quattro condizioni necessarie affinché si verifichi;
- 4 ignorare il problema.**

## 1 Individuare ed 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:

- 1 manuale:** è richiesto all'operatore di intervenire per risolvere la situazione a favore di uno dei processi in modo da sbloccare il sistema;
- 2 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:

- A 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;
- B prerilascio di una risorsa (preemption):** viene forzato il prerilascio di una **risorsa** da parte di uno dei **processi** in **stallo** in modo che questa possa essere acquisita da un **processo** che, grazie a quella allocazione, possa evolvere; questa operazione, però, non può essere effettuata per tutti i tipi di **risorse** (si pensi per esempio a una stampante);
- C checkpoint/rollback:** viene fatto periodicamente il salvataggio su disco dello stato dei **processi** in determinati istanti ben definiti (**checkpoint**): 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**.

## 2 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  $\langle P_1, P_2, \dots, P_n \rangle$  è **sicura** se per ogni  $P_i$  le richieste di risorse che  $P_i$  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.

### Una possibile soluzione: l'algoritmo del banchiere

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.

### 3 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?

Con la prevention il deadlock viene eliminato strutturalmente.

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.

### 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'in disponibilità di una risorsa ad alta priorità ritarda processi che già detengono risorse ad alta priorità.

## 4 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** e da molti sistemi per basi di dati (**ORACLE**, **DB2**, **Informix**, **MySQL**).

Nei **sistemi transazionali**, dato che per motivi di efficienza non è possibile effettuare una transazione alla volta, cioè serializzare gli accessi, il **controllo di concorrenza** avviene in modo "**pessimistico**", cioè vengono assegnati dei **lock** sui dati:

- i **read-lock** possono essere **condivisi**;
- i **write-lock** sono **esclusivi**.

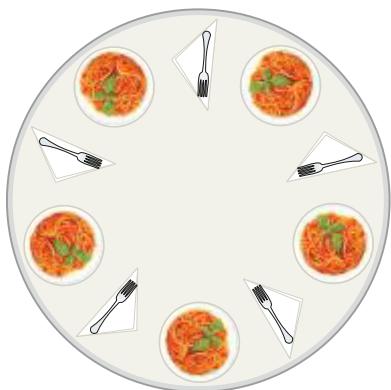
I **lock** concessi sono memorizzati nella **tabella dei lock**: lo studio dettagliato della realizzazione della "transazione ben formata" rispetto al locking e delle anomalie di aggiornamento viene affrontato nei corsi di "**progetto dei database**".

## ■ 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!

# 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.

Un **monitor**:

- ▷ protegge i dati da accessi poco strutturati;
- ▷ garantisce che i dati condivisi siano elaborati solo attraverso interfacce ben definite.

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** (o **thread**) alla volta può essere attivo entro il **monitor** e può quindi richiamare queste procedure (in questo caso si dice che il **processo** è nel **monitor**).

Un **thread** che richiama una procedura di **monitor**, quando un altro **thread** è già nel **monitor**, viene bloccato su un'opportuna coda associata al **monitor** e quando un **thread** all'interno del **monitor** si blocca, un altro **thread** deve poter accedere al **monitor**.

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 entry2()          /* 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).

In un **monitor** è attiva una sola procedura alla volta e questa proprietà è garantita dai meccanismi del supporto a tempo di esecuzione del linguaggio di programmazione corrente e il codice necessario è inserito dal compilatore direttamente nel programma eseguibile.

## ■ 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 pubblici) 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 “nativi” 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 Monitor utilizzato come semaforo

Utilizziamo il **monitor** come un **semaforo**, che permette o meno l'acceso a una **risorsa**.

All'interno del monitor sono presenti i dati condivisi che servono alle entry per regolare l'accesso a:

- ▶ una **variabile booleana** che rappresenta lo stato (libero, occupato) della risorsa;
- ▶ una **variabile condition** per gestire la coda di attesa.

Le funzioni (entry) **richiedi()** e **rilascia()** sono utilizzate solo per garantire l'accesso esclusivo alla risorsa da parte dei processi e la **mutua esclusione** tra le operazioni **richiedi()** e **rilascia()** fa in modo che lo stato della risorsa venga esaminato in modo mutualmente esclusivo dai processi.

Solo se un processo ottiene l'accesso tramite la **entry** può utilizzare la risorsa che è “fuori dal monitor” e, al termine del suo utilizzo, sempre tramite una entry, la rende disponibile agli altri processi.

Una pseudocodifica è la seguente:

```

monitor allocarisorsa
    boolean occupato = false      // variabile privata del monitor (semaforo)
    condition libero           // variabile condition
    procedura entry richiedi()
        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 rilascia()
        inizio
            occupato = false     // se il semaforo è verde
            libero.signal        // risveglio il processo in coda
        fine
    fine monitor

```

Osserviamo come la variabile **occupato** venga utilizzata come un **semaforo**, la entry **richiedi()** equivale alla esecuzione di un **P()** mentre la entry **rilascia()** a quella di un **V()**: l'accesso esclusivo viene però 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**.

Scriviamo ora il segmento di codice che “utilizza la risorsa”

```

allocarisorsa miaRisorsa;           // istanza del tipo monitor
...
    miaRisorsa.richiedi();         // se nessuno usa la risorsa
    < uso della risorsa>;          // elaborazione in mutua esclusione
    miaRisorsa.rilascia();        // risvegli eventuali proc. in coda
...

```

Vediamo un secondo esempio di utilizzo del **monitor** come gestore di una **risorsa**.

### **ESEMPIO 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 **come il C** che non hanno il costrutto **monitor** questo è possibile "costruirlo" mediante **semafori**, anche se non sempre la soluzione è di semplice realizzazione: per ogni istanza di **monitor** il compilatore deve assegnare un **semaforo mutex** per garantire 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 utilizzarsi come 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
```

Vedremo la loro completa implementazione in linguaggio C e alcuni esempi di utilizzo nelle apposite lezioni 6 e 7 di laboratorio.

# ESERCITAZIONI DI LABORATORIO 2

## THREAD E SCHEDULAZIONE

### ■ Un risultato non atteso

In questa lezione vedremo un esercizio che offre lo spunto per fare una riflessione sulla schedulazione dei **processi** e dei **thread** ed evidenzia come sia necessaria la **sincronizzazione**.

L'obiettivo che ci proponiamo è quello di incrementare una variabile condivisa chiamata **globale** sia all'interno del codice del programma principale che all'interno di un **thread** da lui creato: scriviamo un programma in modo che entrambi la incrementino di una unità per 20 volte.

Nel codice, dopo aver incluso le solite librerie, definiamo la variabile **globale** che verrà modificata all'interno del **main** e del **thread**.

```
contatore1.c
1 #include <pthread.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 int globale;
```

Il programma che viene eseguito dal **thread** fa alcune operazioni locali, tra le quali anche quella di sospendersi per un secondo (e ne scopriremo la motivazione in seguito), prima di aggiornare il valore della variabile **globale**:

```
7 void *cosaFaThread(void *arg) {
8     int i, locale;
9     for (i = 0; i < 20; i++) {
10         locale = globale; /* lettura variabile globale */
11         locale = locale + 1;
12         printf(".");
13         fflush(stdout);
14         sleep(1);
15         globale = locale; /* incremento variabile comune */
16     }
17     return NULL;
18 }
```

Il **main()**, dopo aver definito il **thread**, lo crea con la funzione di riga 23 passandogli come parametro il codice che deve eseguire e quindi inizia a incrementare la variabile **globale**, introducendo una pausa dopo ogni incremento.

Sia il **main()** che il **thread** visualizzano sullo schermo un "simbolo" che permette all'utente di identificare la loro esecuzione, così da poterne osservare la loro **schedulazione**.

```
19 int main(void) {
20     pthread_t mioThread;
21     int i;
22     if (pthread_create( &mioThread, NULL, cosaFaThread, NULL) ) {
23         printf("errore nella creazione del thread.");
24         abort();
25     }
26     for (i = 0; i < 20; i++) {
27         globale = globale+i;
28         printf("o"); /* "marca" di incremento fatto dal main */
29         fflush(stdout);
30         sleep(1);
31     }
32     if (pthread_join (mioThread, NULL)){
33         printf("errore nel join del thread.");
34         abort();
35     }
36     printf("\nglobale vale %d\n", globale);
37     exit(0);
38 }
```

Ora osserviamo alcune esecuzioni del programma riportate nella seguente immagine:

```
Paolo@PCwin8 ~/ua2/c_semafori
$ gcc contatore1.c -o conta

Paolo@PCwin8 ~/ua2/c_semafori
$ ./conta
0.0.0.0..00..00..0.0.0..0.0.00.0..00.
globale vale 21

Paolo@PCwin8 ~/ua2/c_semafori
$ ./conta
0.0.0..0.0.0.0..0.00.0.0..0.00.0.0.0..
globale vale 22

Paolo@PCwin8 ~/ua2/c_semafori
$ ./conta
0.0.0..0..00..0.0..00..0.00..0.00.0..0
globale vale 21

Paolo@PCwin8 ~/ua2/c_semafori
$
```

Si può vedere che ogni esecuzione è diversa dalla precedente e che ciascuna fornisce un risultato finale errato, cioè il valore finale di `globale` è pari a 21, 22 oppure anche 23 contro il risultato atteso di 40!



## *Prova adesso!*

- Schedulazione dei thread
  - Condivisione di memoria

Esegui ora lo stesso programma sul tuo calcolatore in modo da avere una esecuzione dello stesso codice su di una macchina diversa: otterrai sempre sequenze di caratteri diverse per ciascuna esecuzione del programma.

Modifica i valori delle istruzioni di sleep(x) e prova a discutere i risultati inattesi ottenuti.

## ■ Problemi di schedulazione e di sincronizzazione

Osservando i risultati dell'esempio precedente possiamo individuare due tipologie di problemi:

- Ⓐ problemi dovuti alla **sincronizzazione**;
- Ⓑ problemi dovuti alla **schedulazione**.

Possiamo affermare che i problemi dovuti alla **sincronizzazione** sono la causa del risultato errato che si presenta sullo schermo: la variabile **globale** nel **main()** viene incrementata direttamente mentre nel **thread** questo avviene utilizzando una variabile **locale**, cioè dopo aver fatto “alcune istruzioni” inserite volutamente per generare un ritardo: il nuovo valore non viene generato direttamente sulla variabile globale ma questa, prima viene copiata in una variabile locale, quindi incrementata e, infine, dopo una pausa di 1 secondo, assegnata alla variabile **globale** che si **sovrappone** al valore che essa ha in quel momento, valore che nel frattempo è stato modificato dal programma principale.

Scrivendo programmi concorrenti che sfruttano i **thread** è doveroso evitare inutili effetti collaterali simili a quello appena visto: cosa si può fare per eliminare questo problema?

Effettuiamo un primo intervento: dato che il problema si verifica perché **globale** è stata copiata in **locale** possiamo provare a evitare l'uso di una variabile **locale** e “operare” direttamente la variabile **globale** nel **thread**

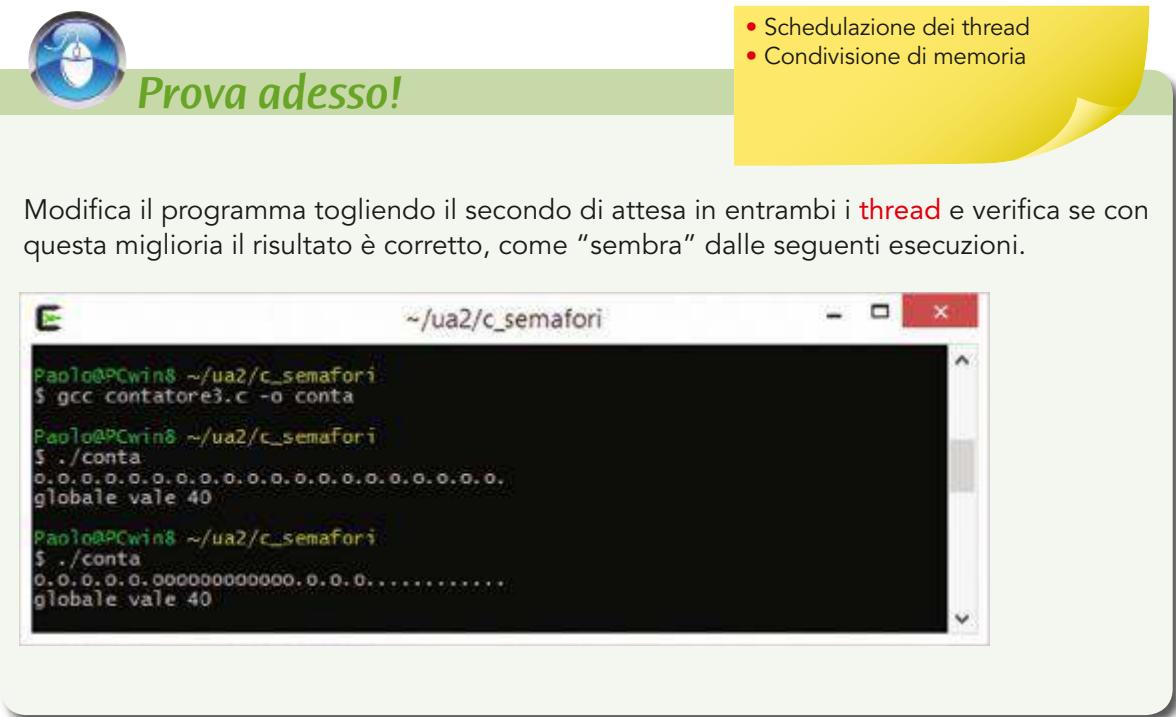
```
7  void *cosaFaThread(void *arg) {  
8      int i;  
9      for (i = 0; i < 20; i++) {  
10          globale = globale + 1;  
11          printf(".");           /* "marca" di incremento fatto dal thread */  
12          fflush(stdout);  
13          sleep(1);  
14      }  
15      return NULL;  
16  }
```

ma i risultati, anche se sono diversi, non producono il valore atteso di 40!



```
Paolo@PCwin8 ~/ua2/c_semafori  
$ gcc contatore2.c -o conta  
Paolo@PCwin8 ~/ua2/c_semafori  
$ ./conta  
o.o.o..o.o.o.o.o.o..oo.o..oo.o.o.  
globale vale 38  
  
Paolo@PCwin8 ~/ua2/c_semafori  
$ ./conta  
o.o.o..o.o.o.o..oo..oo.o..o.o..oo.o.o.  
globale vale 39  
  
Paolo@PCwin8 ~/ua2/c_semafori  
$ ./conta  
o..o.o..o..o..oo..oo..o..o..o..o..o..oo.  
globale vale 38  
  
Paolo@PCwin8 ~/ua2/c_semafori  
$ |
```

Per giustificare questa situazione è necessario ricordare che i **thread** vanno in esecuzione in maniera simultanea e stiamo lavorando su di una macchina uniprocessore, dove il **kernel** realizza il **multitasking** assegnando volta per volta **time slice** di **CPU** ai diversi **processi**: possiamo immaginare che entrambi siamo in esecuzione contemporaneamente e mentre un **thread** “si riposa” per un secondo nel frattempo la variabile condivisa viene modificata dall’altro **thread**.



Con queste ultime rettifiche otteniamo il risultato “apparentemente corretto” se mandiamo in esecuzione il programma su macchine lente: ma anche su di esse, se appena si aumenta la complessità dell’operazione eseguita dal **thread** al posto del semplice incremento di una unità di una variabile, il problema si ripresenterà: lo si può verificare inserendo una semplice moltiplicazione (operazione di riga 12) come sotto riportato (oppure aggiungendo qualche ulteriore operazione algebrica a seconda della velocità del proprio processore).

```
7  void *cosaFaThread(void *arg) {
8      int i, x, y;
9      for (i = 0; i < 20; i++) {
10          globale = globale + 1;
11          x = globale; /* inutili, solo per rallentare l'esecuzione */
12          y = globale * x;
13          printf(".");
14          fflush(stdout);
15      }
16      return NULL;
17 }
```

Per ottenere un risultato “sempre corretto” è necessario introdurre alcuni accorgimenti in modo che un **thread** comunichi all’altro di “bloccarsi” fintanto che non termina di effettuare tutte le operazioni sulle variabili condivise: è quindi necessaria la **sincronizzazione esplicita**.



## Prova adesso!

- Schedulazione dei thread
- Condivisione di memoria

Scrivi un programma che crea due **thread** “**somma1**” e “**somma2**” che accedono alle variabili **datoA** e **datoB** di una struttura dati globale **condivisa** incrementandole rispettivamente di una unità per 5 volte e 10 volte.

Il valore finale delle due variabili verrà stampato a video dal **main()** che per prima cosa definisce e crea le due istanze del **thread**, passandogli come parametro la funzione che devono eseguire: il **main()** rimane quindi in attesa che entrambi i **thread** terminino la loro esecuzione mediante la funzione **pthread\_join()** e come ultime operazioni visualizza sullo schermo i valori finali che assumono le due variabili.

```
somma1Sol.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define VOLTE 5           // costante usata per incrementare le variabili
4
5 struct dati {           // struttura Condivisa fra i thread
6     int datoA;
7     int datoB;
8 } condivisa;
```

Confronta il tuo codice con quello presente nel file **somma1Sol.c**.

Dato che anche in questo esercizio l’accesso alla struttura non viene regolamentato da alcun meccanismo di gestione della concorrenza, cosa ti aspetti come output?

Aggiungi un nuovo **thread** e manda in esecuzione il programma con tre **thread** in contemporanea.

Quindi modifica il codice delle operazioni eseguite da ciascun **thread** introducendo qualche operazione che richiede un maggior tempo di elaborazione (esempio una divisione oppure una radice quadrata): verifica che il nuovo output può avere “dei problemi”.

Come nell’esercizio precedente è necessario introdurre alcuni accorgimenti per fare in modo che un **thread** comunichi a tutti gli altri di “bloccarsi” fintanto che non abbia terminato di effettuare tutte le operazioni sulle variabili condivise: la soluzione avviene introducendo i **mutex** che permettono di gestire la **sincronizzazione**.

# ESERCITAZIONI DI LABORATORIO 3

## I SEMAFORI BINARI IN C



### Info

I codici seguenti sono reperibili nel file **C\_semafori.rar** scaricabile dalla cartella **materiali** nella sezione del sito [www.hoepliscuola.it](http://www.hoepliscuola.it) riservata al presente volume.

### ■ Realizzazione dei semafori binari

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** in attesa di “avere il verde”.

Il **semaforo** deve essere inizializzato prima di essere usato ed è possibile farlo in due modalità.

#### 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.

Le possibili forme di inizializzazione statica di un **mutex** sono:

- **PTHREAD\_MUTEX\_INITIALIZER**: il semaforo parte dallo stato **occupato**;
- **PTHREAD\_RECURSIVE\_MUTEX\_INITIALIZER\_NP**: crea un semaforo binario ricorsivo, ripetendo il tentativo di accesso alla sezione critica per 'n' volte, per cui poi dovremo fare l'unlock 'n' volte;
- **PTHREAD\_ERRORCHECK\_MUTEX\_INITIALIZER\_NP**: se il semaforo è bloccato ritorna il codice di errore EDEADLK
- **PTHREAD\_ADAPTIVE\_MUTEX\_INITIALIZER\_NP**: per i "fast mutex".

Nella nostra trattazione noi utilizzeremo solo la prima costante, cioè inizializzeremo sempre il **semaforo** rosso contemporaneamente alla sua creazione.

## 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 di inizializzazione necessita di due parametri:

- ▶ un **semaforo**, cioè un puntatore a una variabile di tipo **semaforo**;
- ▶ gli attributi per l'inizializzazione del **semaforo**, non necessari per il settaggio di default (parametro **NULL**).

### ESEMPIO 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.

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.

### ESEMPIO Incremento di una variabile condivisa

Scriviamo un programma C in cui una variabile globale **count** viene incrementata di uno per 30 volte da un **thread** creato dal programma e poi incrementata di uno per 20 volte all'interno del **main()**.

Come meccanismo di gestione della **concorrenza** utilizziamo un **mutex** per garantire la “**mutua esclusione per sezione critica**”.

Dopo aver incluso le librerie e le costanti, definiamo le variabili globali che sono:

- ▶ una variabile intera **condivisa** che contiene il dato;
- ▶ il semaforo **mutex** che ci permette di regolare gli accessi a **condivisa**.

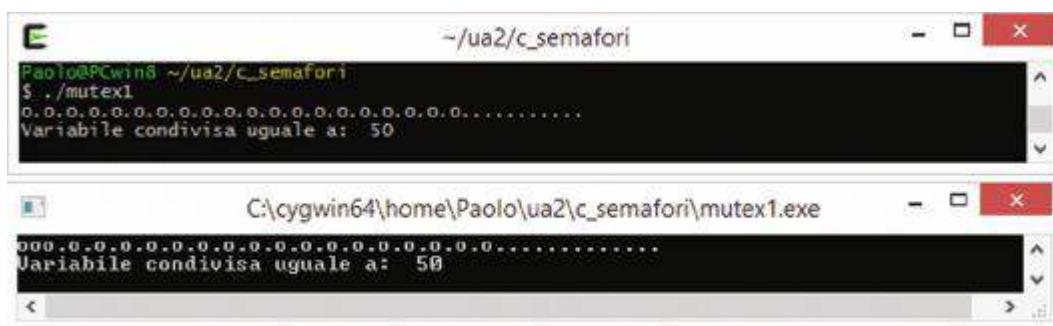
```
mutex1.c

1 #include <pthread.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 # define TANTI1 30
5 # define TANTI2 20
6
7 int condivisa = 0;           // variabile globale condivisa da main e thread
8 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // definizione semaforo
9
10 void *cod_thread(void *arg){   // funzione che viene eseguita dal thread
11     int x, dato;
12     for(x = 0; x < TANTI1; x++) {
13         pthread_mutex_lock(&mutex);    // entra nella sez. critica
14         dato = condivisa;             // aggiorna la variabile condivisa
15         dato++;
16         condivisa = dato;            // copiandola prima in var. locale
17         // solo per rallentare l'esecuzione
18         printf(".");
19         fflush(stdout);
20         pthread_mutex_unlock(&mutex); // esce dalla sez. critica
21     }
}
```

Il **main()** crea il **thread** e successivamente compete con esso per l'aggiornamento della variabile **condivisa**:

```
23 int main(void) {
24     pthread_t tid1;
25     int x, err;
26     if((err = pthread_create(&tid1, NULL, cod_thread, NULL)) != 0) {
27         printf("errore nella creazione thread: %s\n", strerror(err));
28         exit(1);
29     }
30     for(x = 0; x < TANTI2; x++) {
31         pthread_mutex_lock(&mutex); // entra dalla sez. critica
32         condivisa++; // aggiorna direttamente condivisa
33         printf("o");
34         fflush(stdout);
35         pthread_mutex_unlock(&mutex); // esce dalla sez. critica
36     }
37     if(pthread_join(tid1,NULL)) {
38         printf("errore: %s\n",strerror(err));
39         exit(1);
40     }
41     printf("\nVariabile condivisa uguale a: % d\n",condivisa);
42     exit(0);
43 }
```

Riportiamo una esecuzione in Cygwin e una esecuzione in Dev-cpp:



Scriviamo ora un secondo esempio dove vengono creati due **thread** che aggiornano una variabile condivisa finché questa non raggiunge il valore 10: per modificare le sequenze di esecuzioni introduciamo nei due **thread** un ritardo casuale, come si può vedere nella istruzione di riga 16.

```
mutex2.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #define TANTI 10
5 pthread_mutex_t mutex; // semaforo condiviso tra i threads
6 int buffer = 0; // variabile condivisa
7
8 void *cod_thread1 (void *arg) {
9     int accessi1 = 0; // num. di accessi del thread 1 alla sez critica
10    while(buffer < TANTI) { // fino a TANTI
11        pthread_mutex_lock(&mutex); // accesso sez. critica
12        accessi1++;
13        buffer++;
14        printf("accessi di T1: %d valore buffer: %d \n", accessi1, buffer);
15        pthread_mutex_unlock(&mutex); // rilascio sec. critica
16        sleep ((int) (5.0 * rand()/(RAND_MAX + 1.0))); // riposo casuale
17    }
18    pthread_exit (0);
19 }
```

Il codice del secondo **thread** è praticamente identico, tranne che le righe 9 e 14, mentre il **main()** si limita alla creazione dei **thread** e alla attesa della loro terminazione.

```
34 main(){
35     pthread_t tid1, tid2;
36     pthread_mutex_init (&mutex, NULL); // semaforo iniz. à verde
37
38     if (pthread_create(&tid1, NULL, cod_thread1, NULL) < 0) {
39         fprintf (stderr, "errore nella creazione del thread 1\n");
40         exit (1);
41     }
42     if (pthread_create(&tid2, NULL, cod_thread2, NULL) < 0) {
43         fprintf (stderr, "errore nella creazione del thread 2\n");
44         exit (1);
45     }
46     pthread_join (tid1, NULL);
47     pthread_join (tid2, NULL);
48 }
```

Una possibile esecuzione produce il seguente output:

```
Paolo@PCwin8 ~/ua2/c_semafori
$ ./mu2
accessi di T1: 1 valore buffer: 1
accessi di T2: 1 valore buffer: 2
accessi di T1: 2 valore buffer: 3
accessi di T2: 2 valore buffer: 4
accessi di T2: 3 valore buffer: 5
accessi di T1: 3 valore buffer: 6
accessi di T1: 4 valore buffer: 7
accessi di T2: 4 valore buffer: 8
accessi di T2: 5 valore buffer: 9
accessi di T1: 5 valore buffer: 10
Paolo@PCwin8 ~/ua2/c_semafori
$
```



## Prova adesso!

- Utilizzo dei mutex
- Condivisione dei dati in memoria

Scrivi un programma C che crei due **thread** "somma1" e "somma2" che accedono entrambi alle variabili **test.a** e **test.b** di una struttura dati **test** condivisa incrementandole di 1 per 10 volte (20 volte).

Il valore finale delle due variabili verrà stampato a video dal main.

Realizzare come meccanismo di gestione della **concorrenza** quello della "**mutua esclusione per struttura**", dove la struttura **mutex** va allocata **dinamicamente**.

Confronta la tua soluzione con quella presente nel file **mutex3.c**

Scrivi un programma che crea due **thread** che dopo un intervallo di tempo casuale aggiornano metà elementi di un array condiviso inserendo il proprio **pid**.

Si rende necessario l'utilizzo di un lock in modo da avere l'accesso esclusivo all'array e l'aggiornamento dell'indice dell'array. Confronta la tua soluzione con quella presente nel file **mutex4.c**

# ESERCITAZIONI DI LABORATORIO 4

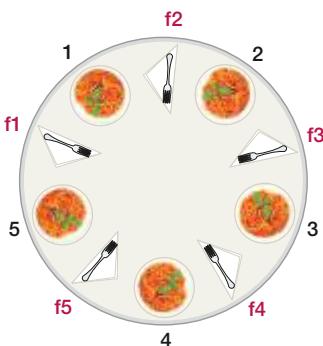
## LA SOLUZIONE DEL DEADLOCK DEI FILOSOFI IN C CON I MUTEX

### ■ Filosofi e deadlock

Si vuole realizzare in linguaggio C la soluzione del problema dei 5 filosofi utilizzando i **semafori binari (mutex)**. Ciascun filosofo, numerato con un valore da 1 a N, può compiere tre attività:

- ▷ pensare
- ▷ mangiare
- ▷ aspettare, nel caso non trovasse disponibili le forchette, numerate per ciascun filosofo:
  - Ⓐ forchetta di destra =  $(x + 1) \% N$
  - Ⓑ forchetta di sinistra =  $(x - 1 + N) \% N$

Schematicamente nel caso di  $n = 5$  filosofi abbiamo la situazione rappresentata in figura.



Associamo a questi tre stati dei valori costanti e indichiamo inoltre il filosofo di destra e quello di sinistra sempre con le due costanti DESTRA e SINISTA, che verranno comode nel seguito del programma sia per individuare i filosofi adiacenti che le posate da utilizzare:

```

5 #define N 5
6 #define PENSA 0
7 #define ATTESA 1
8 #define MANGIA 2
9 #define DESTRA (x + 1) % N
10 #define SINISTRA (x - 1 + N) % N

```

I dati condivisi sono le variabili che indicano lo stato dei filosofi: vengono memorizzate in un vettore e a ciascuna di esse si associa un semaforo per potervi accedere in mutua esclusione:

```
12  pthread_mutex_t mutex, mutex_f[N]; // semafori per RC e forchette
13  int stato[N];                  // variabile condivisa con lo stato dei filosofi
```

Le due attività prevalenti dei filosofi sono quelle di pensare e mangiare; le simuliamo con due funzioni che fanno compiere queste azioni per una durata casuale:

```
15  void pensa(int x){
16      printf("\nFILOSOFO %d: sto pensando ... ", x);
17      sleep(rand() % N);                                // pensa per un tempo casuale tra 0 e N
18  }
19
20  void mangia(int x){
21      printf("\nFILOSOFO %d: sto mangiando ... ", x);
22      sleep(rand() % N);                                // mangia per un tempo casuale tra 0 e N
23  }
```

Per sapere se il generico filosofo  $x$ -esimo può mangiare è necessario osservare lo stato dei filosofi adiacenti: 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 il semaforo associato al filosofo mettendolo a rosso quando inizia a mangiare e rilasciandolo quando termina di mangiare.

Quando il filosofo termina di mangiare richiamiamo la seguente funzione:

```
25  void posa(int x){
26      pthread_mutex_lock(&mutex);                      // accesso in mutua esclusione
27      stato[x] = PENSA;                                // aggiorna il suo stato
28      pthread_mutex_unlock(&mutex_f[x]);                // libera la sua risorsa
29      pthread_mutex_unlock(&mutex);                    // rilascia la regione critica
30  }
```

Quando invece vuole iniziare a mangiare richiamiamo una funzione con una struttura simile alla seguente:

```
39  void prendi(int x){
40      pthread_mutex_lock(&mutex);                      // accesso in mutua esclusione
41      printf("\nFILOSOFO %d: ho fame e aspetto le forchette ... ", x);
42      stato[x] = ATTESA;
43      controlloPosate(x);                            // controlla e attende le posate
44      printf("\nFILOSOFO %d: ...ora prendo le forchette e mangio", x);
45      pthread_mutex_lock(&mutex_f[x]);                // occupa la sua risorsa
46      pthread_mutex_unlock(&mutex);                  // rilascia la regione critica
47  }
```

La funzione `controlloPosate(x)` effettua i controlli sulle posate libere e si comporta di conseguenza.



## Prova adesso!

Realizza la funzione controlloPosate(x) in modo da garantire il corretto funzionamento in assenza di starvation e deadlock; per simulare la "vita del filosofo" utilizzando la seguente funzione:

```

49 void *filosofo(void *x){
50     int k = (int) x;
51     while(1){
52         pensa(k);
53         prendi(k);
54         mangia(k);
55         posa(k);
56     }
57 }
```

Il sistema viene mandato in esecuzione dal seguente `main()`:

```

59 void main(){
60     int x;
61     srand((int) time(NULL));           // inizializza il seme x random
62     pthread_t filo[N];               // definizione dei thread
63     pthread_mutex_init(&mutex, NULL); // inizializzazione semafori
64     for(x = 0; x < N; x++)
65         pthread_mutex_init(&mutex_f[x], NULL);
66     for(x = 0; x < N; x++){
67         pthread_create(&filo[x], NULL, (void *)filosofo, (void *)x); // crea i thread
68         sleep(1);
69     }
70 }
```

Confronta la tua soluzione con quella presente nel file [filosofi.c](#)

Infine modifica il numero di filosofi e manda in esecuzione il programma: cosa puoi osservare?

# ESERCITAZIONI DI LABORATORIO 5

## LA SOLUZIONE DEL PROBLEMA PRODUTTORI/CONSUMATORI CON I SEMAFORI CLASSICI

### ■ Semafori in C

La prima versione di **POSIX** metteva a disposizione del programmatore solo due primitive per la sincronizzazione dei **thread** in processi multipli: **mutex** e le **variabili condizione**.

Con lo standard **IEEE POSIX 1003.1b** (1993) vengono introdotti come estensione real-time anche i **semafori classici**, mediante l'inclusione dell'**header**:

```
#include <semaphore.h>
```

In questa libreria sono presenti molteplici funzioni che operano su un nuovo tipo di dato, il **semaforo**

```
sem_t <nome semaforo>
```

I **semafori** presenti in **POSIX1.b** possono essere **named** e **unnamed**: noi utilizzeremo i **semafori unnamed** che possiedono funzioni simili ai **mutex** e per utilizzarli in applicazioni multiprocesso basta allocarli nella **memoria condivisa**.

Descriviamo sinteticamente le principali operazioni su variabili di tipo **sem\_t**.

- 1 Per inizializzare un **semaforo unnamed** si usa la funzione:

```
int sem_init(sem_t *sem, int pshared, unsigned int valore )
```

Dove:

**sem\_t \*sem** è il puntatore al semaforo da inizializzare;

**int pshared** nell'attuale implementazione deve essere posto a 0; in implementazioni future potrà essere posto a 1 nel caso che il **semaforo** dovrà essere condiviso tra più processi.

L'effetto di questa istruzione è quello di inizializzare il semaforo **sem** con il valore **valore**: come al solito avrà valore di ritorno 0 in caso di successo e -1 in caso di fallimento: deve sempre essere fatta dal programmatore.

- 2 Per attendere indefinitamente l'**acquisizione** di un semaforo, cioè per effettuare l'operazione che esegue una **P(sem)**, si ha a disposizione la funzione:

```
int sem_wait(sem_t *sem)      // equivale alla operazione di P(sem)
```

Il funzionamento della **sem\_wait()** è quello della **P()** di Dijkstra: il semaforo può essere considerato come un intero e la funzione **sem\_wait()** esegue un test per verificarne il valore:

- ▶ se il semaforo è minore o **uguale a 0** (**semaforo rosso**), la **sem\_wait()** si sospende in coda, forzando un cambio di contesto a favore di un **processo pronto**;
- ▶ se il semaforo presenta un valore maggiore o **uguale a 1** (**semaforo verde**), la **sem\_wait()** decrementa tale valore e ritorna al chiamante, che può quindi procedere nella sua elaborazione.

- 3 Per tentare di **acquisire** un semaforo ma continuare l'evoluzione se questo è rosso, esiste la funzione "non bloccante":

```
int sem_trywait (sem_t *sem)
```

particolarmente utile nelle situazioni dove potrebbero esserci possibili deadlock.

- 4 Per **rilasciare** un semaforo, cioè per eseguire un segue una **V(sem)**, si usa:

```
int sem_post (sem_t *sem)
```

L'operazione di **sem\_post()** incrementa il contatore del **semaforo** e se dopo tale operazione il valore risulta ancora minore od **uguale a zero** significa che altri processi hanno effettuato la **wait\_sem()** ma hanno trovato il semaforo rosso e sono stati posti in attesa: quindi si sveglia il primo in coda.

- 5 Per controllare il valore di un **semaforo** si usa la funzione:

```
int sem_getvalue (sem_t *sem, int *sem_val);
```

Il valore del semaforo viene ritornato nella variabile **\*sem\_val**

- 6 Per distruggere un semaforo si usa la funzione:

```
int sem_destroy (sem_t *sem)
```

**AREA digitale**



Nota per gli utenti MAC

**ESEMPIO**

Vediamo un primo esempio dove riscriviamo il codice che risolve il problema del **produttore/consumatore**, in particolare dove un **produttore** e un **consumatore** si alternano per un numero di **VOLTE** nella scrittura/prelievo di un dato da **buffer** comune.

```
semProCon1.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5 #define VOLTE 5
6 // memoria condivisa
7 static sem_t mio_sem;
8 int buffer = 0; // var. condivisa con il dato
9 int conta = 0; // var. condivisa per nr ripetizioni
```

Il **thread produttore** esegue la **P()** sul **semaforo** ed entra nella sezione critica dove controlla se è il suo turno (**buffer vuoto**) e in tal caso scrive un numero progressivo (**conta + 1**) nel **buffer**; quindi esce dalla sezione critica eseguendo la **V()**:

```
11 void *produc1 (void * arg){
12     int avanti = 1; // variabile di ctr del ciclo
13     while(avanti) {
14         sem_wait (&mio_sem); // P() sul semaforo mio_sem
15         if (conta < VOLTE){
16             if (buffer == 0){ // se il dato è stato consumato
17                 buffer = conta + 1; // aggiorna il dato per il lettore
18                 printf("T1: scritto %d \n", buffer);
19             };
20         } else {
21             avanti = 0; // fine ripetizione del ciclo
22             printf("T1 ha finito \n");
23         }
24         sem_post(&mio_sem); // V() sul semaforo mio_sem
25     }
26     pthread_exit (0);
27 }
```

Il **thread consumatore** esegue la **P()** sul **semaforo** ed entra nella sezione critica dove controlla se è il suo turno (**buffer pieno**) e in tal caso legge (e consuma mettendolo uguale a 0) il dato presente nel **buffer**; quindi esce dalla sezione critica eseguendo la **V()**:

```
30 void *consum1 (void * arg){
31     int avanti = 1; // variabile di ctr del ciclo
32     while(avanti){
33         sem_wait (&mio_sem); // P() sul semaforo mio_sem
34         if (conta < VOLTE){
35             if (buffer > 0){
36                 printf("T2: letto %d \n", buffer);
37                 conta = buffer; // incremento le volte
38                 buffer = 0; // "consumo" il dato
39             };
40         } else {
41             avanti = 0; // fine ripetizione VOLTE del ciclo
42             printf("T2 ha finito \n");
43         }
44         sem_post(&mio_sem); // V() sul semaforo mio_sem
45     }
46     pthread_exit (0);
47 }
```

Il `main()` inizializza il semaforo `mio_sem` a verde, crea i due `thread` e si pone in attesa della loro terminazione.

```

50 void main () {
51     pthread_t tid1, tid2;
52     sem_init(&mio_sem, 0, 1);           // all'inizio semaforo verde
53
54     if (pthread_create(&tid1, NULL, produci, NULL) < 0){
55         fprintf (stderr, "errore nella creazione di thread 1\n");
56         exit (1);
57     }
58     if (pthread_create(&tid2, NULL, consumi, NULL) < 0) {
59         fprintf (stderr, "errore nella creazione di thread 2\n");
60         exit (1);
61     }
62     pthread_join (tid1, NULL);
63     pthread_join (tid2, NULL);
64 }
```

Compiliamo e mandiamo in esecuzione il programma ottenendo la seguente schermata:

```

Paolo@PCwin8 ~/ua2/c_semafori
$ gcc semProCon1.c -o pc1
Paolo@PCwin8 ~/ua2/c_semafori
$ ./pc1
T1: scritto 1
T2: letto 1
T1: scritto 2
T2: letto 2
T1: scritto 3
T2: letto 3
T1: scritto 4
T2: letto 4
T1: scritto 5
T2: letto 5
T1 ha finito
T2 ha finito
Paolo@PCwin8 ~/ua2/c_semafori
$
```



## Prova adesso!

- Semafori binari
- Un produttore – Due consumatori

- 1 Scrivi un programma dove un produttore scrive tante VOLTE un carattere sempre diverso in un buffer e 2 consumatori a turno lo leggono e lo visualizzano facendo in modo che il produttore inserisca un nuovo carattere solo dopo che entrambi i consumatori lo hanno letto.
- 2 Confronta la tua soluzione con quella riportata nel file [sem1Prod2Cons.c](#)



## Prova adesso!

- Semafori binari
- Un produttore – N consumatori

- 1 Scrivi un programma dove un produttore scrive tante VOLTE un carattere sempre diverso in un buffer e N consumatori a turno lo leggono e lo visualizzano: il produttore può produrre un nuovo carattere solo dopo che almeno uno dei consumatori ha letto quello precedente.
- 2 Confronta la tua soluzione con quella riportata nel file [sem1ProdNConsChar.c](#)

## AREA digitale



Soluzione del problema del produttore/consumatore con i mutex

## ■ Semafori per la gestione di vincoli di precedenza

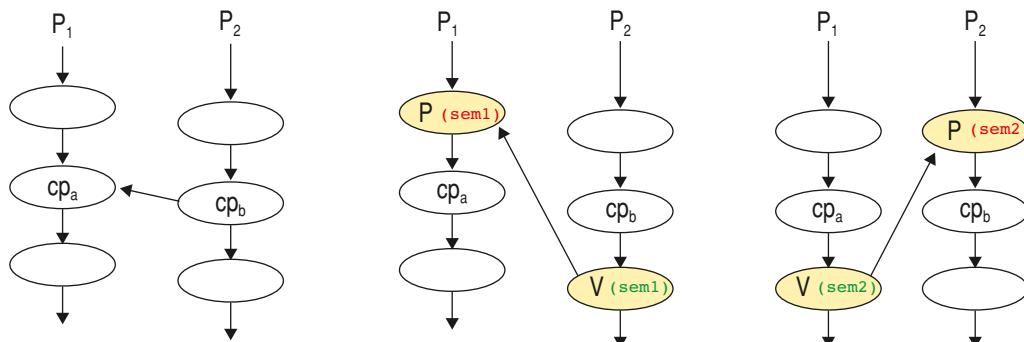
La soluzione del problema [produttore/consumatore](#) ottenuta con un solo [semaforo](#) concede a tutti i [processi](#) di accedere al [buffer](#) senza vincoli temporali, cioè i [processi consumatori](#) possono accedere al [buffer](#) anche se nessun [produttore](#) è presente nel sistema: per come abbiamo implementato il meccanismo di [sincronizzazione](#) questi [processi consumatori](#) consumano risorse per niente in quanto prima di sapere se è il loro turno accedono alla sezione critica per effettuare il controllo sul buffer in modo da accertarsi se questo è pieno/vuoto e si regolano di conseguenza.

Questa strategia non è sicuramente funzionale in quanto occupiamo la [sezione critica](#) inutilmente al pari di una [attesa attiva](#): facendo una analogia nella vita quotidiana, è come se noi andiamo continuamente a guardare nella casella di posta per vedere se è arrivato un documento.

Miglioriamo l'efficienza del sistema introducendo un vincolo temporale, cioè permettendo l'accesso alla regione critica ai [consumatori](#) solo dopo che un [produttore](#) ha inserito un dato. Per realizzare questo meccanismo inseriamo nel sistema [due semafori](#):

- un primo [semaforo sem1](#) associato “agli scrittori” che inizializziamo a zero (verde);
- un secondo [semaforo sem2](#) per i lettori che inizializziamo a 1 (rosso).

In riferimento allo schema seguente:



la sequenza delle operazioni è:

- ▷ un produttore P1 prima di eseguire  $op_a$ , esegue  $P(sem1)$  e blocca altri produttori: accede in mutua esclusione alla SC (semaforo rosso per gli scrittori);
- ▷ dopo aver eseguito  $op_a$ , P1 esegue  $V(sem2)$ : risveglia i consumatori "lasciando dormire" gli altri produttori;
- ▷ un consumatore P2 prima di eseguire  $op_b$ , esegue  $P(sem2)$  e blocca altri consumatori: accede in mutua esclusione alla SC;
- ▷ dopo aver eseguito  $op_b$ , P2 esegue  $V(sem2)$ : vengono risvegliati i produttori "lasciando dormire" gli altri consumatori.

Riassumendo, per realizzare il meccanismo di **mutua esclusione** utilizzando due **semafori** abbiamo:

- ▷ **producì**: semaforo che è verde quando la memoria non contiene dati utili e quindi il **produttore** può scrivere in essa;
- ▷ **leggi**: semaforo che è verde quando la memoria contiene dati utili e quindi il **produttore** può accedere per la lettura.

Lo schema generale di **sincronizzazione** è il seguente:

```
semaforo produci = verde;
semaforo leggi    = rosso;
produttore {
    P(producì);
    ... produci ...
    V( leggi)
}
consumatore {
    P(leggi);
    ... consuma ...
    V(producì);
}
```

Nel nostro esempio facciamo eseguire dieci volte la produzione e il consumo del dato: questo si limita a essere un numero intero che contiene proprio il contatore delle iterazioni effettuate.

Dato che il **produttore** testa il **semaforo** verde, sarà il primo ad accedere alla regione condivisa ed a depositare un dato: finito di scrivere sveglierà il **consumatore** modificando il valore del semaforo **leggi** al quale è associata la lista di attesa dei **consumatori** e solo allora il primo **processo** in coda potrà quindi accedere alla **regione critica**, inizierà a leggere il dato e alla fine della lettura metterà a verde il **semaforo** di **producì**, permettendo al **produttore** di inserire un nuovo dato... e così via per dieci iterazioni.

In questo primo esempio creiamo un solo **produttore** e un solo **consumatore**: vedremo nel seguito come completare la casistica aumentando sia il numero dei **produttori** che dei **consumatori**.

La codifica in C richiede l'importazione di quattro librerie, la definizione dei due **semafori** e la definizione delle variabili condivise:

```
semProCon2.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5 #define MAX 5
6
7 static sem_t produci, leggi;
8 int buffer = 0; // variabile condivisa con il dato
9 int volte = 0; // variabile condivisa per nr ripetizioni
```

Il codice dei **thread produttore** non richiede spiegazioni aggiuntive in quanto è simile alla versione precedente tranne che per le istruzioni 14 e 23 dove vengono settati i due **semafori**:

```
14 sem_wait(&produc); // P() sul semaforo produci dei produttori
15 if (volte < MAX){
16     buffer = volte + 1;
17     printf("T1: scritto %d \n", buffer);
18 }
19 else{
20     avanti = 0; // fine ripetizione del ciclo
21     printf("T1 ha finito \n");
22 }
23 sem_post(&leggi); // V() sul semaforo leggi dei consumatori
```

Analogo discorso per il **thread consumatore** per le istruzioni 31 e 40.

```
31 sem_wait(&leggi); // P() sul semaforo leggi dei consumatori
32 if (volte < MAX){
33     printf("T2: letto %d \n", buffer);
34     volte = buffer; // incremento le volte
35 }
36 else {
37     avanti = 0; // fine ripetizione del ciclo
38     printf("T2 ha finito \n");
39 }
40 sem_post(&produc); // V() sul semaforo produci dei produttori
```

Riportiamo la prima parte del **main()** dove vengono inizializzati i semafori dei quali quello che indica il dato pronto viene inizializzato staticamente a **rosso**:

```
44
45 int main () {
46     pthread_t tid1, tid2;
47     sem_init(&produc, 0, 1); // verde per la produzione
48     sem_init(&leggi, 0, 0); // rosso per la consumazione
49 }
```

Compiliamo e mandiamo in esecuzione il programma ottenendo lo stesso output del primo esempio, ma in questa realizzazione i **processi** non consumano inutilmente tempo **CPU!**

**AREA** *digitale*



Mutex per la gestione di vincoli di precedenza



## Prova adesso!

- Semafori sem\_t
- Utilizzo di P() e V()
- Sincronizzazione

- 1 Scrivi un programma dove un thread1 esegua un insieme di operazioni (chiamate fase 1) e al suo termine risvegli un thread2 che a sua volta esegue un insieme di operazione (chiamate fase2) regolando la loro sequenzialità mediante un **semaforo**.
- 2 Confronta la tua soluzione con quella presente nel file **semSequenza1.c**
- 3 Scrivi un programma dove tre processi ciclicamente incrementano a turno una variabile rispettando la sequenza P1 -> P2 -> P3
- 4 Confronta la tua soluzione con quella presente nel file **semCiclico1.c**
- 5 Scrivi un programma dove due processi ciclicamente "giocano a ping-pong" rilanciandosi per tante VOLTE la pallina sincronizzandosi con due semafori, sem1 e sem2.
- 6 Confronta la tua soluzione con quella presente nel file **semPingPong1.c**
- 7 Scrivi 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".
- 8 Confronta la tua soluzione con quella presente nel file **semCampane1.c**

## ■ N produttori e N consumatori

Il problema dei **produttori/consumatori** nel caso più generale può essere così formulato: alcuni processi **produttori** utilizzando un buffer di scambio devono continuamente inviare messaggi ad altri processi **consumatori** che li elaborano nello stesso ordine in cui li ricevono:

- i **produttori** scrivono i messaggi nel buffer, uno alla volta;
- i **consumatori** leggono i messaggi dal buffer, ciascuno un messaggio diverso.

I due tipi di processi devono essere opportunamente sincronizzati in modo **da evitare** queste situazioni:

- un **consumatore** legge un messaggio senza che un **produttore** ne abbia depositato alcuno;
- un **produttore** sovrascrive un messaggio scritto prima che un **consumatore** sia riuscito a leggerlo;
- un **messaggio** viene letto più di una volta dai **consumatori**.

Scriviamo un codice parametrico in modo da poter personalizzare con delle costanti:

- il numero dei produttori: **NUM\_THREAD\_P**;
- il numero dei consumatori: **NUM\_THREAD\_C**;
- il numero di messaggi che devono essere prodotti: **VOLTE**.

Per la sincronizzazione utilizziamo tre **semafori**:

- un **MUTEX** per regolare l'accesso ai dati condivisi;
- un **semaforo vuoto** per regolare i **produttori**;
- un **semaforo pieno** per regolare i **consumatori**.

```

semMutexPC.c

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5
6 #define VOLTE 33 // elementi da produrre
7 #define NUM_THREAD_P 3 // numero produttori
8 #define NUM_THREAD_C 3 // numero consumatori
9 #define FALSO 0
10 #define VERO 1
11
12 char buffer = 'a'-1; // scritto dai produttori e letto dai consumatori
13 int scritti = 0; // modificato dai produttori
14 int letti = 0; // modificato dai consumatori
15 int fine = FALSO; // modificato dai produttori quando hanno prodotto VOLTE
16
17 sem_t pieno, vuoto; // semafori per la sincronizzazione
18 pthread_mutex_t mutex; // semaforo per la sezione critica

```

Il codice del **thread produttore** è costituito da:

- A** un ciclo a condizione finale che ripete la produzione degli elementi così strutturato;
  - una istruzione genera casualmente un tempo di attesa che “simula il tempo in cui un **produttore** produce il suo prodotto”;
  - l’istruzione di **P(vuoto)** che regola l’accesso nel caso in cui il buffer sia disponibile per “inserire” la produzione oppure sospende il **thread** in attesa della **wait** (istruzione 26);
  - il controllo del **mutex** che garantisce la mutua esclusione alle risorse condivise;
  - il test per verificare se è necessario produrre nuovi messaggi oppure i **produttori** hanno “finito il loro lavoro”;
  - la produzione del messaggio, che si limita a incrementare di una unità il carattere presente nel **buffer** e ad aggiornare il contatore dei messaggi prodotti;
  - viene infine risvegliata la coda dei **consumatori** con la **V(pieno)** con la istruzione 38 e rilasciato il **mutex** della regione critica con l’istruzione 39;
- B** dalla coda viene risvegliato eventualmente un **thread** produttore in attesa in modo da cauterarsi per quelle situazioni in cui alla fine di tutta la produzione ci fossero altri produttori in attesa di un evento che non accadrebbe mai: quindi prima di terminare la sua vita un produttore deve risvegliare un “collega produttore” affinché possa anch’esso terminare (istruzione 46).

```

20 void *produci(void *threadid){
21     long tid;
22     tid = (long) threadid;
23     printf("\n Ciao a tutti: sono il thread produci #%ld!", tid);
24     do{
25         sleep(rand() % 3); // prepara la "produzione"
26         sem_wait(&vuoto); // blocca altri produttori
27         // -----
28         pthread_mutex_lock(&mutex); // entra nella sez. critica
29         if (scritti < VOLTE){
30             buffer++; // nuovo dato prodotto - condiviso
31             scritti++; // contatore produzione -condiviso
32             printf("\n produci #%ld %d", tid, scritti);
33             fflush(stdout);
34         }
35         else{
36             fine = VERO; // scrive su memoria condivisa fine produzione
37         }
38         sem_post(&pieno); // risveglia i consumatori
39         pthread_mutex_unlock(&mutex); // esce dalla sez. critica
40         // -----
41     }while(scritti < VOLTE); // manca ancora produzione
42
43     // ---parte finale ---
44     printf("\n produci #%ld ha finito ", tid);
45     fflush(stdout);
46     sem_post(&vuoto); // necessario se C < P per far terminare i produttori
47     return NULL;
48 }

```

Il codice del **consumatore** è molto simile dove viene dapprima testato il **semaforo** per verificare se il buffer è pieno e di seguito testato il **mutex** che regola la sezione critica.

Per meglio controllare l'evoluzione del programma introduciamo alcune istruzioni di output che visualizzano il contenuto delle variabili.

```

49 void *consuma(void *threadid){
50     long tid;
51     tid = (long) threadid;
52     printf("\n Ciao a tutti: sono il thread consuma #Xld!", tid);
53     char miobuffer = '\x0';
54     do{
55         sleep(rand() % 3);           // si riposa ...
56         sem_wait(&pieno);        // blocca altri consumatori
57         //...
58         pthread_mutex_lock(&mutex); // entra nella sez. critica
59         if (letti < scritti){
60             miobuffer = buffer;    // legge variabile condivisa
61             letti++;              // modifica variabile condivisa
62             printf("\n consuma #Xld ha letto Xc: %d di %d", tid, miobuffer, letti, scritti);
63             fflush(stdout);
64         }
65         pthread_mutex_unlock(&mutex); // esce dalla sez. critica
66         sem_post(&vuoto);          // risveglia i produttori
67         //...
68     } while ((fine == FALSE) ); // non prodotti tutti o non consumati tutti
69     // ---parte finale
70     printf("\n consuma #Xld ha finito ", tid);
71     fflush(stdout);
72     sem_post(&pieno);          // necessario se P < C
73     return NULL;               // sem_post(&vuoto); // necessario se C < P
74 }
```

Dopo aver definito due array per memorizzare i tid dei **thread produttori** e **consumatori**, il **main()** inizializza i **semafori** ponendo a verde quelle dei produttori ed a rosso quello dei **consumatori**. Quindi avvia tutti i **thread** e si mette in attesa della loro terminazione.

```

76 int main(void){
77     pthread_t thread_P[NUM_THREAD_P]; // vettore dei produttori
78     pthread_t thread_C[NUM_THREAD_C]; // vettore dei consumatori
79     long tc, tp ;
80     printf("\nInizio elaborazione...");
81     sem_init(&pieno, 0, 0);           // inizia a falso (rosso)
82     sem_init(&vuoto, 0, 1);          // inizia a vero (verde)
83     for (tp = 0; tp < NUM_THREAD_P; tp++) {
84         if (pthread_create(thread_P+tp, NULL, produci, (void *)tp+1) < 0){
85             fprintf (stderr, "errore nella creazione del thread P%d\n", tc);
86             exit (1);
87         }
88     }
89     for (tc = 0; tc < NUM_THREAD_C; tc++){
90         if (pthread_create(thread_C+tc, NULL, consuma, (void *)tc+1) < 0){
91             fprintf (stderr, "errore nella creazione del thread C%d\n", tc);
92             exit (1);
93         }
94     }
95     for (tp = 0; tp < NUM_THREAD_P; tp++) // attesa terminazione produttori
96         pthread_join(thread_P[tp], NULL);
97     for (tc = 0; tc < NUM_THREAD_C; tc++) // attesa terminazione onsumatori
98         pthread_join(thread_C[tc], NULL); // bisognerebbe distruggere i 3 semafori
99     printf("\nFine elaborazione !");    // prima di terminare l'elaborazione
100 }
```

Una esecuzione con 2 produttori, 4 consumatori e 7 messaggi da produrre dà il seguente output: ►

```

Inizio elaborazione...
Ciao a tutti: sono il thread produci #2!
Ciao a tutti: sono il thread produci #1!
Ciao a tutti: sono il thread consuma #1!
Ciao a tutti: sono il thread consuma #2!
Ciao a tutti: sono il thread consuma #3!
Ciao a tutti: sono il thread consuma #4!
    produci #2 1
    consuma #4 ha letto a: 1 di 1
    produci #1 2
    consuma #2 ha letto b: 2 di 2
    produci #1 3
    consuma #1 ha letto c: 3 di 3
    produci #2 4
    consuma #3 ha letto d: 4 di 4
    produci #2 5
    consuma #2 ha letto e: 5 di 5
    produci #1 6
    consuma #4 ha letto f: 6 di 6
    produci #1 7
    produci #1 ha finito
    consuma #3 ha letto g: 7 di 7
    produci #2 ha finito
    consuma #1 ha finito
    consuma #3 ha finito
    consuma #4 ha finito
    consuma #2 ha finito
Fine elaborazione !

```



## Prova adesso!

- N produttori – N consumatori
- Buffer circolare
- Lettori e scrittori

- 1 Scrivi un programma dove NUM\_THREAD\_P produttori scrivono tante VOLTE un carattere sempre diverso in un buffer circolare di dimensione DIMBUFFER e NUM\_THREAD\_C consumatori a turno lo leggono e lo visualizzano.  
Confronta la tua soluzione con quella riportata nel file [semMutexCirco.c](#). È possibile utilizzare un unico semaforo? Discuti la soluzione proposta nel file [semMutexCirco1.c](#)
- 2 Quattro amici fanno una scommessa, il premio per il vincitore è di poter fare bere uno dei rimanenti amici. Chi vince la scommessa sceglie casualmente a chi tocca bere il cocktail tutto d'un fiato mentre gli altri stanno a guardare. Alla fine, viene proposta un'ulteriore scommessa e il gioco va avanti all'infinito. Rappresentare il problema utilizzando i semafori.
- 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.

L'uso di **semafori** a livello di programma è comunque ostico e rischioso:

- il posizionamento improprio delle P può causare situazioni di blocco infinito (**deadlock**) o anche esecuzioni erronee di difficile verifica (**race condition**);
- è indesiderabile lasciare all'utente il pieno controllo di strutture così delicate;
- linguaggi evoluti di alto livello offrono strutture **esplicite** di controllo delle regioni critiche come i **monitor**.



# ESERCITAZIONI DI LABORATORIO 6

## VARIABILI CONDIZIONE

### ■ Pthread condition variable

Il linguaggio C ha nella libreria `<pthread.h>` oltre ai **semafori** descritti nelle lezioni precedenti un ulteriore strumento adatto per “sospendere” i **thread** in attesa della verifica di situazioni particolari: le **variabili condizione** (**condition variable**).

Le **variabili condizione** sono un meccanismo che viene utilizzato per sospendere l'esecuzione di un **thread** in attesa che si verifichi un certo evento.

Le **variabili condizione** sono molto diverse dai **semafori di sincronizzazione**, anche se semanticamente fanno la stessa cosa: a ogni **condition** viene associata una **coda** per la sospensione dei **thread** ma la **variabile condizione** non ha uno **stato** (libero o occupato), rappresenta solo una **coda** nella quale i **thread** possono essere sospesi e, quindi, le operazioni fondamentali sono la **sospensione e la riattivazione** che vengono effettuate tramite due primitive (**wait** e **signal**).

Le primitive delle **condition** si preoccupano di rilasciare e acquisire la **mutua esclusione** prima di bloccarsi e dopo essere state sbloccate, ma una **variabile condizione** non fornisce la **mutua esclusione**: ha sempre bisogno di un **mutex** per poter **sincronizzare** l'accesso ai dati mentre i **semafori** non necessitano della presenza di altri meccanismi.

Lo schema di funzionamento della **sincronizzazione** è il seguente: abbiamo detto che una **variabile condition** è **sempre** associata a un **mutex** quindi la prima operazione che esegue il **thread** è quella di ottenere il **mutex** e di testare il predicato:

- se il predicato è verificato allora il **thread** esegue le sue operazioni e rilascia il **mutex**;
- se il predicato non è verificato, in modo atomico, il **mutex** viene rilasciato (implicitamente) e il **thread** si blocca sulla **variabile condition**.

Successivamente un **thread** bloccato riacquisisce il **mutex** nel momento in cui viene svegliato da un altro **thread**.

Linux garantisce inoltre che i **thread** bloccati su una **condition** vengano sbloccati quando questa cambia di stato.

## ■ Definizione di una variabile condizione

Attraverso le **variabili condizione** è però possibile implementare condizioni più complesse che i **thread** devono soddisfare per essere eseguiti: oltre che a essere un nuovo tipo di variabile (**pthread\_cond\_t**) hanno anche attributi di un nuovo tipo (**pthread\_condattr\_t**).

È possibile effettuare la creazione di una **variabile condizione** e la contestuale inizializzazione con quella che prende il nome di **inizializzazione statica**.

### Inizializzazione statica di una variabile condizione

Una **variabile condizione** è creata e inizializzata mediante la seguente istruzione simile a quella già utilizzata per i **mutex**:

```
pthread_cond_t miavc = PTHREAD_COND_INITIALIZER;
```

dove

► **miavc** è il nome di una **variabile condizione**;

► **PTHREAD\_COND\_INITIALIZER** è una macro di inizializzazione definita nella libreria **<pthread.h>** che setta gli attributi di **miavc** con i valori di default.

### Inizializzazione dinamica di una variabile condizione

È anche possibile inizializzare una **variabile condizione** in fase run-time mediante la seguente istruzione:

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr)
```

dove:

```
pthread_cond_t *cond
```

puntatore a un'istanza della variabile **condition** (condizione di sincronizzazione):

```
pthread_condattr_t *cond_attr
```

puntatore a una struttura che contiene gli attributi della condizione: può anche assumere il valore **NULL** e in questo caso viene inizializzato con i valori di default.

### Distruzione di una variabile condizione

Al termine della elaborazione è necessario deallocare lo spazio in memoria, così come abbiamo visto per i **semaphori**: viene richiamata la seguente funzione:

```
int pthread_cond_destroy(pthread_cond_t *cond)
```

Affinché la funzione dia un esito positivo e ritorni il valore 0 non devono essere presenti **thread** in attesa sulla variabile **cond** altrimenti viene ritornato un valore diverso da 0.

## ■ Primitive fondamentali: wait e signal

Le primitive fondamentali sulle **variabile condizione** sono due: **wait()** per la **sospensione** e **signal()** per la **riattivazione**.

### Attesa su una variabile condizione: wait

Per mettersi in attesa di una **certa condizione** all'interno di un blocco di dati condivisi e protetti da un **mutex** un **thread** utilizza la seguente **system call**:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
```

dove i parametri sono:

```
pthread_cond_t *cond
```

è il puntatore all'istanza di **condition** che rappresenta la condizione di sincronizzazione:

```
pthread_mutex_t *mutex
```

è l'indirizzo di un **semaforo** di mutua esclusione necessario alla corretta consistenza dei dati regolando l'accesso alla regione critica agli altri **thread** dopo aver messo nella coda di attesa il **thread** corrente.

La funzione ha sempre 0 come valore di ritorno.

Alla chiamata di **wait()** il **mutex** viene sbloccato mentre il **thread** chiamante **si blocca sempre** sulla **variabile condition**, in attesa di essere risvegliato da un segnale (una successiva **signal()**).

La presenza del **mutex** fra i parametri garantisce che, al momento del bloccaggio del **thread** che esegue la **wait()**, esso venga posto automaticamente a verde, eliminando a monte possibili errori di programmazione che potrebbero condurre a condizioni di **deadlock**.

Quando il **thread** sospeso riceve un segnale di risveglio, entra in competizione per accedere a bloccare il **mutex** prima di riprendere a eseguire le successive istruzioni della **sezione critica**.

Per effettuare una **wait()** e quindi attendere su una **variabile condition** un **thread** deve aver precedentemente effettuato una **lock()** sul **mutex** e non sulla **variabile condition**, che **non è** un "oggetto booleano":

```
if (miaCV)
    then ...
if (risorseLibere == 0)
    then wait(&miaCV, &mutex) // è errata e senza significato
                                // è formalmente corretta
```

## Risveglio di una variabile condizione: signal

La **signal()** non si preoccupa di liberare la **mutua esclusione**: fra i suoi parametri, infatti, non c'è il **mutex** ma solo la **variabile condition**, quindi il **mutex** deve essere rilasciato esplicitamente dal programmatore con una successiva istruzione altrimenti si potrebbe produrre una condizione di **deadlock**.

Una **variabile condizione** può essere svegliata in due modi: **standard** oppure **broadcast**.

### a) Standard: sblocca un solo thread bloccato

La modalità standard, che è quella più utilizzata, sblocca il **thread** a priorità più alta che è in attesa da più tempo.

```
int pthread_cond_signal (pthread_cond_t *cond)
```

Se esistono **thread** sospesi nella coda associata a **cond** ne viene risvegliato il primo ma se non vi sono **thread** in attesa sulla condizione la **signal()** non produce alcun effetto e viene persa. Come unico parametro ha

```
pthread_cond_t *cond
```

che è il puntatore all'istanza di **condition** che rappresenta la condizione di sincronizzazione. La funzione ha sempre 0 come valore di ritorno.

### b) Broadcast: sblocca tutti i thread bloccati

Con la chiamata alla seguente funzione:

```
int pthread_cond_broadcast(pthread_cond_t *cond)
```

vengono sbloccati tutti i **thread** sospesi sulla variabile **condition** rispettando l'ordine di arrivo **FIFO** in caso siano presenti **thread** con medesima priorità.

Come unico parametro ha

```
pthread_cond_t *cond
```

che è il puntatore all'istanza di **condition** che rappresenta la condizione di sincronizzazione. La funzione ha sempre 0 come valore di ritorno.

Ne è sconsigliato l'utilizzo dato che il **thread** svegliato deve rivalutare la condizione in quanto l'altro **thread** potrebbe non aver testato la condizione e nel frattempo questa potrebbe essere stata cambiata.

## Osservazioni sul risveglio

Quando un **thread A** effettua una **signal()** e risveglia un **thread B**, questo riprenderà la sua attività all'interno della **sezione critica**, poiché è proprio all'interno della **SC** che si è sospeso: ma il **thread A** che lo ha risvegliato non ha ancora lasciato la **SC** quindi, per un "breve intervallo di tempo", non viene rispettato il **principio della mutua esclusione** all'interno della **SC**, dato che in essa sono presenti contemporaneamente due **thread**.

L'intervallo di tempo di convivenza dipende dall'approccio operativo che viene adottato dal programmatore per il comportamento del **thread A** dopo che ha risvegliato il **thread B**:

- 1 se viene seguito l'approccio di Brinch Hansen il **thread A** viene fatto uscire immediatamente dalla **SC** sbloccando il **mutex**;
- 2 se viene seguito l'approccio di Hoare il **thread A** viene posto in attesa finché il **thread B** svegliato non usa più la **risorsa**.

Generalmente si usa il primo approccio in modo da limitare il più possibile la "convivenza".

## ■ 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);           // accede alla sc
...
pthread_cond_wait(& condVar, &mutex); // accede alla sc
...
pthread_mutex_unlock(&mutex);        // esce dalla sc
```

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(&condVar, &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.

### ESEMPIO Accesso alla SC regolato da condition variable

Definiamo due semafori, un **mutex** e una **condition variable**:

varCondition1.c	
<pre>1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 #include &lt;pthread.h&gt; 4 5 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // mutex 6 pthread_cond_t condVar = PTHREAD_COND_INITIALIZER; // condition variable</pre>	

Il codice dei due **thread** rispecchia lo schema prima descritto:

```

8  void produci(void *parametro) {
9      printf("Avvio della esecuzione del %s.\n", (char *)parametro);
10     sleep(2);           // pausa di comodo
11     printf("Thread 1 tenta di entrare nella sezione critica.\n");
12     pthread_mutex_lock(&mutex);
13     printf("Thread 1 nella sezione critica - ha bloccato il mutex.\n");
14     printf("Thread 1 produce ....\n");
15     sleep(1);           // pausa di comodo
16     printf("Thread 1 si sospende sulla condition variable.\n");
17     pthread_cond_wait(&condVar, &mutex);
18     printf("Thread 1 riprende l'esecuzione.\n");
19     printf("Thread 1 ora esce dalla sezione critica - rilascia mutex\n");
20     pthread_mutex_unlock(&mutex);
21     printf("Thread 1 puo' terminare.\n");
22 }
```

All'avvio il primo **thread** 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**.

```

24 void consuma(void *parametro) {
25     printf(" Avvio della esecuzione del %s.\n", (char *)parametro);
26     sleep(5);           // pausa di comodo
27     printf(" Thread 2 tenta di entrare nella sezione critica.\n");
28     pthread_mutex_lock(&mutex);
29     printf(" Thread 2 nella sezione critica - ha bloccato mutex.\n");
30     printf(" Thread 2 consuma ....\n");
31     sleep(1);           // pausa di comodo
32     printf(" Thread 2 sblocca chi e' in attesa sulla condition variable.\n");
33     pthread_cond_signal(&condVar);
34     printf(" Thread 2 ora esce dalla sezione critica - rilascia mutex.\n");
35     pthread_mutex_unlock(&mutex);
36     printf(" Thread 2 puo' terminare.\n");
37 }
```

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.

Il programma **main** si limita a creare i due **thread**, a mandarli in esecuzione e ad attendere la terminazione:

```

40 void main(){
41     pthread_t thread1, thread2;
42     char *nome1 = "Thread 1";
43     char *nome2 = "Thread 2";
44     if(pthread_create(&thread1, NULL, &producি, nome1) != 0) {
45         perror("Errore nella creazione del primo thread.\n");
46         exit(1);
47     }
48     if(pthread_create(&thread2, NULL, &consumا, (void*)nome2) != 0) {
49         perror("Errore nella creazione del secondo thread.\n");
50         exit(1);
51     }
52     pthread_join(thread1, NULL);
53     pthread_join(thread2, NULL);
54 }
```

Un'esecuzione è la seguente:

```

Avvio della esecuzione del Thread 1.
Avvio della esecuzione del Thread 2.
Thread 1 tenta di entrare nella sezione critica.
Thread 1 nella sezione critica - ha bloccato il mutex.
Thread 1 produce ...
Thread 1 si sospende sulla condition variable.
Thread 2 tenta di entrare nella sezione critica.
Thread 2 nella sezione critica - ha bloccato mutex.
Thread 2 consuma ...
Thread 2 sblocca chi e' in attesa sulla condition variable.
Thread 2 ora esce dalla sezione critica - rilascia mutex.
Thread 2 puo' terminare.
Thread 1 riprende l'esecuzione.
Thread 1 ora esce dalla sezione critica - rilascia mutex
Thread 1 puo' terminare.

```



## Prova adesso!

- Variabile condition
- Sincronizzazione thread

Completa il codice presente nel file **varCondition2.c** dove un gruppo di **thread** eseguono alcune operazioni (da noi simulate con un tempo di attesa random) e al loro termine si pongono tutti in attesa degli altri, cioè aspettano gli altri **thread** su una "barriera" prima di poter proseguire l'elaborazione.

Quando tutti i **thread** sono pronti per proseguire, viene inviato un segnale che ne permette la ripresa della elaborazione.



# ESERCITAZIONI DI LABORATORIO 9

## 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](http://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:

- innanzitutto 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 **x** della **classe InComune1**;
  - passando ai **thread** un reference di un oggetto **Contatore** creato nel processo padre (conta).
- Scriviamo dapprima la **classe Contatore**:

```

1 class Contatore{
2     int valore;
3     int passo;
4     Contatore(int valore, int passo){
5         this.valore = valore;
6         this.passo = passo;
7         System.out.println("\nIl contatore e' nato e vale " + this.getValore());
8     }
9     void incrementa(){
10         valore += passo;
11     }
12     void decrementa(){
13         valore -= passo;
14     }
15     int getValore(){
16         return valore;
17     }
18 }
```

Nel costruttore del **Contatore** è stata inserita l'istruzione 7 `println()` unicamente per visualizzare dove viene creata e inizializzata la variabile **valore** (istruzione 33 del thread `main()`).

Proprio con l'istruzione 33 del metodo `main()` creiamo un oggetto **Contatore** e passiamo ad entrambi i `thread` il suo reference in modo che possano condividerlo:

```

1 public class InComune{
2     // si ambiente globale: visibili da ogni thread
3     protected static int x = 100;
4
5     public static void main(String [] args){
6         // oggetto comune passato al thread
7         Contatore conta = new Contatore(0,1);
8         Thread thr1 = new UnThread1("Yogi",conta);
9         Thread thr2 = new UnThread1("Bubù",conta);
10        thr1.start();
11        thr2.start();
12    }
13 }
```

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.incrementa()` (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;           // variabili interne al servizio inizializzate dal costruttore
    private int delta = 1;
    private String nickname = "-";   // nickname
    Contatore conta;               // costruttore
    // esecuzione
    UnThread1(String nomethread, Contatore conta){
        this.nickname = nomethread;
        this.conta = conta;
    }
    public void run() {
        for(; ;){                   // ripeti per sempre
            System.out.println(nickname+":inizia='"+inizia+", x='"+InComune.x+"', contatore='"+conta.getValore()+"')");
            inizia += delta;
            InComune.x++;           // modifica direttamente le variabili della classe che lo crea (consigliato)
            conta.incrementa();    // modifica ok
            try {Thread.sleep(1500);}
            catch (InterruptedException e){System.out.println(e);}
            if(inizia > 5)           // termina dopo 5 ripetizioni
                return;
        }
    }
}
```

```

Il contatore e' nato e vale 0
Bubù:iniziala=0, x=100, contatore=0
Yogi:iniziala=0, x=100, contatore=0
Yogi:iniziala=1, x=102, contatore=2
Bubù:iniziala=1, x=102, contatore=2
Yogi:iniziala=2, x=104, contatore=4
Bubù:iniziala=2, x=104, contatore=4
Yogi:iniziala=3, x=106, contatore=6
Bubù:iniziala=3, x=106, contatore=6
Yogi:iniziala=4, x=108, contatore=8
Bubù:iniziala=4, x=108, contatore=8
Yogi:iniziala=5, x=110, contatore=10
Bubù:iniziala=5, x=110, contatore=10

```

Avviandone una esecuzione otteniamo il seguente output: ►

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** ad ogni iterazione di ciascun **thread** “subiscono” un incremento, cioè sono *effettivamente comuni ai due thread*.

L’output può però anche mostrare la presenza di 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.

## ■ Il qualificatore **synchronized**

Nei **thread** la risorsa condivisa è costituita dalle variabili globali e quindi implementiamo le primitive per accedere ad esse e utilizzarle. 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**: ogni istanza di una classe thread-safe (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 metodo1(...) {
        // sezione critica
        ...
    }
    ...
    public synchronized void metodo2(...) {
        // sezione critica
        ...
    }
    ...
}
```

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, cioè a un singolo blocco di istruzioni, mediante il costrutto:

```
public void metodo3(...){
    // sezione non critica
    ...
    synchronized(this){
        // 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**.

## ■ Realizzazione dei semafori in Java

Java non ammette **semafori** esplicativi come elementi del linguaggio, ma proprio per la sua natura di linguaggio a oggetti permette di costruire una classe che realizza i **semafori di Dijkstra**.

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 **InterruptedException**;
- 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 tramite la clausola **synchronized**:

```

1 class Semaforo{
2     int valore; // 0 = rosso
3     public Semaforo(int v){
4         valore = v;
5     }
6
7     synchronized public void P(){
8         while (valore == 0){ // semaforo rosso
9             try { wait(); } // il thread si sospende
10            catch(InterruptedException e){}
11        }
12        valore--; // pone il semaforo a rosso
13    } //end P
14
15     synchronized public void V(){
16         valore++; // pone semaforo a verde
17         notify(); // risveglia thread in coda
18    } //end V
19 }
```

Il **costruttore** ci permette di inizializzare il **semaforo** a rosso oppure a verde: utilizziamo questa classe per implementare una situazione tipica di produttore/consumatore.

### ESEMPIO **Un produttore e un consumatore regolato da semaforo**

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 **buffer** che servirà memoria condivisa per i due **thread** e due **semafori** (**pieno**, **vuoto**) 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.

I semafori vengono passati come parametri nel costruttore dei due **thread** nel **main()** della classe di prova: ►

```

1 public class ProdConsSem {
2     protected static int buffer;           // variabile condivisa globale
3     public static void main(String args[]){
4         Semaforo pieno = new Semaforo(0); //inizialmente rosso
5         Semaforo vuoto = new Semaforo(1); //inizialmente verde
6         Producido prod = new Producido(pieno, vuoto);
7         ConsumaDato cons = new ConsumaDato(pieno, vuoto);
8         prod.start();
9         cons.start();
10        System.out.println("Main: termine avvio thread.");
11    }
12 }
```

Il **thread Producido** si limita a scrivere un dato nel buffer quando trova verde il **semaforo** che indica buffer vuoto: altrimenti si sospende e, al termine della operazione di scrittura, setta a verde il **semaforo** che risveglia il **consumatore**.

```

1 class Producido extends Thread{
2     Semaforo pieno;
3     Semaforo vuoto;
4     int tanti = 5;                      // numero di elementi da produrre
5     final int attesa = 500;              // tempo di riposo/attesa
6     public Producido(Semaforo s1, Semaforo s2){
7         pieno = s1;
8         vuoto = s2;
9     }
10
11    public void run(){
12        for (int k = 0; k < tanti; k++){ // per tutti i dati da produrre
13            vuoto.P();
14            ProdConsSem.buffer = k;      // produce un numero
15            System.out.println("Scrittore: dato scritto :" + k);
16            pieno.V();
17            try {Thread.sleep(attesa);}
18            catch (Exception e){ }
19        }
20        System.out.println("Scrittore: termine scrittura dati.");
21    } //fine run
22 }
```

Il **thread ConsumaDato** 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**. ►

```

1 class ConsumaDato extends Thread {
2     Semaforo pieno;
3     Semaforo vuoto;
4     int dato;
5     public ConsumaDato(Semaforo s1, Semaforo s2){
6         pieno = s1;
7         vuoto = s2;
8     }
9
10    public void run() {
11        while (true){
12            pieno.P();
13            dato = ProdConsSem.buffer; // consuma un numero
14            System.out.println("Lettore: dato letto :" + dato);
15            vuoto.V();
16        }
17    } //fine run
18 }
```

Mandando in esecuzione il programma, si ottiene il seguente output: ►

Possiamo osservare che in questa implementazione si verifica una situazione indesiderata: il **thread consumatore** non termina mai!

Sarà quindi necessario "rivedere" la strutturazione del programma.

```

BlueJ Terminal Window - TPSIT2 - UDA2
Options
Main: termine avvio thread.
Scrittore: dato scritto :0
Lettore: dato letto 0
Scrittore: dato scritto :1
Lettore: dato letto 1
Scrittore: dato scritto :2
Lettore: dato letto 2
Scrittore: dato scritto :3
Lettore: dato letto 3
Scrittore: dato scritto :4
Lettore: dato letto 4
Scrittore: termine scrittura dati.

```



## Prova adesso!

- 1 Modifica il programma precedente facendo in modo che il consumatore termini la sua esecuzione dopo che ha letto l'ultimo dato prodotto dal consumatore.
- 2 Modifica il programma utilizzando come buffer un oggetto di tipo contatore utilizzando la classe Contatore definita nel primo esempio.
- 3 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.
- 4 Realizza un programma per sincronizzare un produttore che riempie un buffer 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.
- 5 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.
- 6 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".
- 7 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 11

## 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.

#### Deserzione 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:

```

1 public class CiclistaInGara implements Runnable {
2     Ciclista ciclista;
3     GaraCicistica campo;
4     int velocita;           // numero di pixel di spostamento al secondo: range 1-4
5     Thread t;
6     int conta;             // ogni 10 spostamenti cambio la velocità
7     int posizione;          // coordinate X in espressa in pixel
8
9     public CiclistaInGara(Ciclista c, GaraCicistica g){
10         ciclista = c;           // identificativo del ciclista
11         campo = g;             // campo di gara - pista
12         conta = 0;
13         velocita = 2;
14         t = new Thread(this);
15         t.start();
16         posizione = 0;
17     }
18
19     public void run(){
20         // muove il ciclista lungo il percorso, cambiano la velocità
21         int dimImmagine = 75; // dimensione della immagine del concorrente
22         int dimPista = 860;
23         while((ciclista.getCordX() + dimImmagine) < dimPista){ // corsa non finita
24             if((conta % 10) == 0) {                                // ogni 10 spostamenti
25                 velocita = (int)(Math.random()*4 + 1);           // modifica la velocità
26                 ciclista.setCordX(ciclista.getCordX() + velocita);
27                 conta++;
28                 try(Thread.sleep(75));                         // delay
29                 catch(Exception e){}
30                 campo.repaint();                            // aggiorna la pista
31             }
32
33             // scrivo in che posizione è arrivato nella classifica finale
34             posizione = campo.getPosizione();
35             campo.controlloArrivi();
36         }
37     }
38 }
```

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

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:

```

1 import java.awt.*;
2 import javax.swing.*;
3
4 public class GaraCiclistica extends JFrame{
5     int posizione;
6     Ciclista[] partecipanti;
7     CiclistaInGara[] thread_partecipanti;
8     Campo pista;
9     Graphics offscreen;           // per la gestione del doppio buffering
10    Image buffer_virtuale;
11

```

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

```

12     public GaraCiclistica(){
13         //posiziono e disegno il percorso.
14         super("Gara Ciclistica");
15         setSize(1000,645);
16         setLocation(10,40);
17         setDefaultCloseOperation(EXIT_ON_CLOSE);
18         pista = new Campo();
19         partecipanti= new Ciclista[6];
20         thread_partecipanti= new CiclistaInGara[6];
21         posizione = 1;
22         //aggiungo le immagini dei concorrenti e li associo ai Thread
23         int partenza = 15;           // posizione della prima corsia
24         for(int xx = 0; xx < 6; xx++){
25             partecipanti[xx]= new Ciclista(partenza, xx + 1);
26             thread_partecipanti[xx]= new CiclistaInGara(partecipanti[xx],this);
27             partenza+=100;          // posizione verticale - corsia del ciclista
28         }
29         // visualizzo la gara
30         this.add(pista);
31         setVisible(true);
32     }

```

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;

```

34     public synchronized int getPosizione(){
35         return posizione++;
36     }

```

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

```

37     public synchronized void controllaArrivo(){
38         boolean arrivati = true;
39         for(int xx = 0; xx < 6; xx++){
40             if(thread_partecipanti[xx].posizione == 6){
41                 arrivati = false;
42             }
43         }
44         if(arrivati){
45             visualizzaClassifica();
46         }
47     }

```

La classifica viene realizzata dal seguente metodo:

```

19 public void visualizzaClassifica(){
20     JLabel[] arrivi;
21     arrivi = new JLabel[6];
22     JFrame classifica=new JFrame("Classifica");
23     classifica.setSize(500,500);
24     classifica.setLocation(250,150);
25     classifica.setBackground(Color.BLUE);
26     classifica.setDefaultCloseOperation(EXIT_ON_CLOSE);
27     classifica.getContentPane().setLayout(new GridLayout(6,1));
28     //visualizza l'ordine di arrivo
29     for(int xx = 1; xx < 7; xx++){
30         for(int yy = 0; yy < 6; yy++){
31             if(thread_partecipanti[yy].posizione == xx){
32                 arrivi[yy]=new JLabel(xx + " classificato ciclista in " +(yy+1)+" corsie");
33                 arrivi[yy].setFont(new Font("Times New Roman",Font.ITALIC,20));
34                 arrivi[yy].setForeground(Color.blue);
35                 classifica.getContentPane().add(arrivi[yy]);
36             }
37         }
38     }
39     classifica.setVisible(true);
40 }
```

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:

```

51 public void update(Graphics g){
52     paint(g);
53 }
54
55 public void paint(Graphics g){
56     if (partecipanti != null){
57         Graphics2D screen = (Graphics2D)g;
58         buffer_virtuale = createImage(1000, 645);
59         offscreen = buffer_virtuale.getGraphics();
60         Dimension d = getSize();
61         pista.paint(offscreen);
62         for(int xx = 0; xx < 6; xx++){
63             partecipanti[xx].paint(offscreen);
64         }
65         screen.drawImage(buffer_virtuale, 0, 30, this);
66         offscreen.dispose();
67     }
68 }
```

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

```

81 public static void main(String[] args){
82     GaraCiclistica m = new GaraCiclistica();
83 }
84 }
```

Riportiamo per completezza la classe **Ciclista** che si occupa del caricamento e della assegnazione delle immagini ai rispettivi ciclisti: ▼

```

1 import java.awt.*;
2 import javax.swing.*;
3
4 public class Ciclista extends JPanel{
5     int cordx;
6     int cordy;
7     Image img;
8
9     public Ciclista(int yy,int xx){
10         cordx = 0;
11         cordy = yy;
12         setSize(50, 51);
13         Toolkit tk=Toolkit.getDefaultToolkit();
14         switch (xx){           // ogni ciclista ha la maglia di colore diverso
15             case 1: (img = tk.getImage("bici1.JPG"));break;
16             case 2: (img = tk.getImage("bici2.JPG"));break;
17             case 3: (img = tk.getImage("bici3.JPG"));break;
18             case 4: (img = tk.getImage("bici4.JPG"));break;
19             case 5: (img = tk.getImage("bici5.JPG"));break;
20             case 6: (img = tk.getImage("bici6.JPG"));break;
21         }
22         MediaTracker mt=new MediaTracker(this); // oggetto per la gestione di un num. arbitrario
23         mt.addImage(img, 1);                  // di immagini in parallelo
24         try{mt.waitForID(1);}
25         catch(InterruptedException e){}
26     }
27 }
```

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

```

21 public void setCordx(int n){
22     cordx = n;
23 }
24
25 public int getCordx(){
26     return cordx;
27 }
28
29 public void paint(Graphics g){
30     g.drawImage(img, cordx, cordy, null);
31 }
32 }
```

Completa il programma la classe **Campo** che disegna il “campo di gara”, cioè la pista:

```

1 import javax.swing.*;
2 import java.awt.*;
3
4 public class Campo extends JPanel {
5     public void paint(Graphics g){
6         g.setColor(Color.green);
7         g.fillRect(0, 0, 1000, 645);
8         //linee laterali
9         g.setColor(Color.white);
10        g.fillRect(0, 0, 1000, 10);
11        g.fillRect(0, 100, 1000, 10);
12        g.fillRect(0, 200, 1000, 10);
13        g.fillRect(0, 300, 1000, 10);
14        g.fillRect(0, 400, 1000, 10);
15        g.fillRect(0, 500, 1000, 10);
16        g.fillRect(0, 600, 1000, 10);
17        //taguardo
18        g.fillRect(960, 0, 5, 645);
19        g.fillRect(970, 0, 5, 645);
20        g.fillRect(980, 0, 5, 645);
21    }
22 }
```

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.

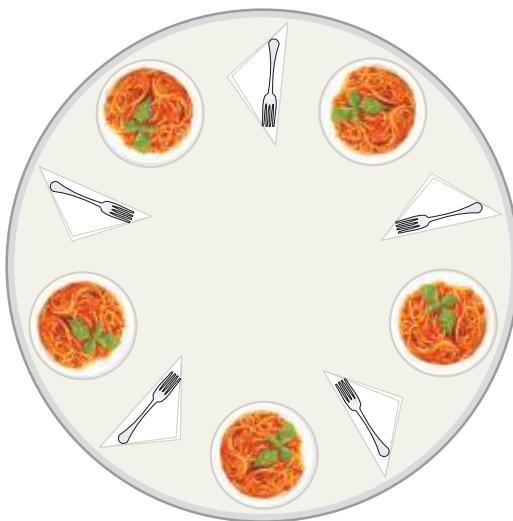
# ESERCITAZIONI DI LABORATORIO 12

## 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:

```

1 class Consuma extends Thread{
2     private Buffer buffer;           // monitor per il dato condiviso
3     private int nome;               // identificativo del consumatore
4     private int tanti;              // num. dei dati da consumare
5     public Consuma(Buffer buffer, int numero, int quanti){
6         this.nome = numero;
7         this.buffer = buffer;
8         this.tanti = quanti;
9     }
10    public void run(){
11        for(int xx = 0; xx < tanti; xx++){
12            int valore = buffer.togli();
13            System.out.println("Consumatore Nr. " + nome+ " prende : " + valore);
14        }
15    }
16 }
```

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 deadlock 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 Tavola** e avvia l'esecuzione dei singoli **thread**:

```

1 public class FilosofiCena{
2     public static void main(String[] args){
3         Filosofo filosofi[];
4         filosofi = new Filosofo[5];
5         Tavola tavola = new Tavola();
6         for(int n = 0; n < 5; n++)           // creazione dei 5 filosofi
7             filosofi[n] = new Filosofo(tavola, n);
8
9         for(int n = 0; n < 5; n++)          // avvio dei 5 filosofi
10            filosofi[n].start();
11    }
12 }
```



## 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 lasciaForchette(int nrFilosofo){
        <rileggi le forchette>
        notifyAll();           // risveglia i filosofi in attesa
    }
}
```

Ora modifica il numero di filosofi e manda in esecuzione il programma: cosa puoi osservare?