Processi

Il multi-tasking

I primi computer permettevano di eseguire un solo programma alla volta, cedendo a esso il controllo completo del sistema. Ciò comportava uno scarso sfruttamento del processore a causa dei tempi morti per l'attesa del completamento delle operazioni di I/O. Della multiprogrammazione, ovvero avere più programmi caricati contemporaneamente in memoria, non venivano dunque sfruttate le intrinseche potenzialità computazionali.
Il passo successivo non poteva che essere quello di utilizzare questi tempi morti di attesa per eseguire più programmi, almeno apparentemente, in parallelo. Il **multi-tasking** garantisce proprio questo, gestendo la turnazione dei programmi sul processore quando il programma in esecuzione è in attesa della risposta delle periferiche. La multiprogrammazione rimane ovviamente una condizione necessaria, e in particolare sarebbe preferibile avere tra tutti i programmi caricati in memoria almeno uno che dia qualcosa da fare al processore, o il multi-tasking non sarebbe sfruttato. Chiaro che ciò non può avvenire sempre, l'importante è che accada il più spesso possibile.

I processi

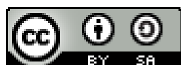
I *programmi in esecuzione* di cui abbiamo appena parlato sono i **processi**, l'unità di lavoro nei moderni sistemi a condivisione di tempo (time-sharing). Sono composti dal **codice del programma** (anche detto *sezione di codice*) e dai **dati del programma**, che a loro volta si suddividono in:

- *variabili globali*, allocate in memoria centrale nell'area dati globali
- *variabili locali e non locali* delle procedure del programma, memorizzate in uno *stack*
- *variabili temporanee introdotte dal compilatore*, tra cui ricordiamo il *program counter*, caricate nei registri del processore
- *variabili allocate dinamicamente* durante l'esecuzione, memorizzate in uno *heap*

Importantissimo sottolineare che **i programmi non sono processi**. I primi sono infatti entità passive, una sequenza di istruzioni contenute in un file sorgente salvato su disco. I processi sono invece entità attive, con un program counter che specifica l'istruzione successiva da eseguire, un gruppo di istruzioni in uso e una particolare istanza dei dati su cui era stato mandato in esecuzione il programma. E' per questo che posso benissimo avere due processi associati allo stesso programma, verranno comunque considerati come istanze di esecuzione distinte dello stesso codice. Ad esempio, posso aprire due finestre di *Firefox* e gestirle separatamente: il loro sorgente è uguale, la sezione dati è differente.
Un altro modo di pensare i processi è come flussi di esecuzione della computazione. Indipendentemente da cosa fanno o rappresentano, possiamo intuitivamente sostenere che se due flussi sono separati, anche i processi lo sono, e che in questo caso potrebbero evolversi coordinandosi (*processi sincronizzati*) o in modo assolutamente autonomo (*processi indipendenti*).
In base ai flussi posso inoltre distinguere due modelli di computazione, il *processo monolitico* e quello *cooperativo*. Nel primo vengono eseguite tutte le istruzioni del programma in un' unico flusso dall'inizio alla fine; mentre nel secondo vengono generati una serie di processi concorrenti che lavorano (a livello logico) in parallelo per conseguire lo stesso scopo. Da notare come i processi monolitici siano tra loro indipendenti, mentre i vari processi che computano a livello cooperativo interagiscono tra loro (ad esempio condividendo i risultati, sincronizzandosi, ecc). La realizzazione di tale modelli di computazione può dunque essere di tre tipi: programma monolitico eseguito come tale, programma monolitico che genera processi cooperanti, programmi separati eseguiti come cooperanti.

Evoluzione della computazione

Lo *stato di evoluzione della computazione* può essere considerata una terza componente dei processi, ed indica a che punto è arrivata la loro esecuzione. Tali stati fotografano istante per istante le istruzioni eseguite e il valore dei dati del processo, permettendoci di prevedere come evolverà il programma semplicemente conoscendo l'istruzione successiva indirizzata.
Si parla di *evoluzione* perché durante l'esecuzione del processo avviene una trasformazione delle informazioni. Lo si può dunque immaginare come una funzione, in cui su valori iniziali vengono eseguite delle operazioni che produrranno un risultato finale, o come una macchina a stati finiti, dove gli stati sono le informazioni su cui opera e le transizioni le istruzioni che li modificano.
Ricapitolando, lo stato di evoluzione della computazione di un processo è l'insieme dei valori di tutte le informazioni da cui dipende l'evoluzione della computazione del processo. Non si limita quindi al solo valore corrente del program counter, ma anche al contenuto delle variabili utilizzate dal programma, dato che una loro eventuale modifica



Questo/a opera è pubblicato sotto una Licenza Creative Commons.

potrebbe portare ad esiti diversi della computazione. Da qui l'importanza che l'esecuzione di un processo non alteri quella di un altro, o riscontrerei evoluzioni diverse da quelle corrette: applicare le stesse operazioni su dati diversi, può non condurre agli stessi risultati.

Stato dei processi

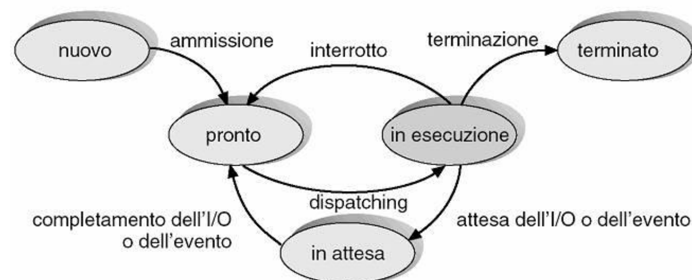
Un processo in esecuzione può assumere diversi stati rispetto all'uso della CPU. Notare bene che non si sta più parlando di stati di computazione, ma *stati di uso del processore da parte di un processo*, o in breve **stati del processo**, che rappresentano la modalità di uso corrente del processore e delle risorse da parte del processo.

Ciascun processo può trovarsi in uno dei seguenti stati:

- **new** (*nuovo*), se è stato appena creato e inizializzato
- **running** (*in esecuzione*), se le istruzioni sono eseguite regolarmente e la computazione evolve effettivamente. In ogni istante, solo un processo per processore può essere in questo stato
- **waiting** (*in attesa*), se il processo sta aspettando il verificarsi di qualche evento, ad esempio che gli vengano assegnate delle risorse o il completamento di un'operazione di I/O
- **ready-to-run** (*pronto all'esecuzione*), se ha tutte le risorse necessarie allo svolgimento delle sue attività, eccetto la CPU da cui aspetta di essere chiamato. La computazione potrebbe dunque evolvere, ma non lo fa perché le istuzioni non possono essere eseguite dal processore
- **terminated** (*terminato*), se ha terminato l'esecuzione e sta aspettando che il sistema operativo rilasci le risorse che utilizzava e lo rimuova dalla memoria (il che non avviene sempre immediatamente).

I nomi dei vari stati non sono definiti univocamente e infatti possono cambiare a seconda del sistema operativo; ciò che rappresentano è invece comune a tutti i sistemi, che al più potranno introdurne altri.

Il diagramma degli stati dei processo è un grafo orientato che rappresenta l'insieme degli stati del processo (i nodi) e le transizioni tra essi (gli archi).



Process control block

Il sistema operativo rappresenta i processi in una struttura dati nota come **process control block** (PCB, *blocco di controllo del processo*) o *task control block* in cui ne vengono memorizzate le principali informazioni. Tra queste, che potrebbero variare a seconda dei sistemi, ricordiamo:

- *identificatore del processo*, un numero unico e univoco
- *stato del processo*, considerato in questo caso come stato di uso del processore da parte del processo
- *program counter*
- *registri della CPU*, che variano per numero e tipo a seconda dell'architettura del computer. Sono ad esempio i registri indice e altri registri di uso generico
- lo *stack pointer*
- *informazioni per la schedulazione della CPU*, che specifica quali tecniche e criteri di schedulazione dovrà operare il processore sul processo
- *informazioni per la gestione della memoria centrale*
- *informazioni per l'accounting*
- *informazioni sullo stato dell'I/O*, che riportano la lista dei file e delle periferiche associate al processo

Non tutte le informazioni sono strettamente necessarie come supporto della gestione del processore, ma vengono comunque inglobate per avere un'unica tabella omogenea a livello di contenuto.

Le PCB relative a ogni singolo processo si riveleranno fondamentali nell'implementazione del multi-tasking.

Cambio di contesto

Questo/a opera è pubblicato sotto una Licenza Creative Commons.

Se il mio obiettivo è realizzare un sistema multi-tasking, far passare la CPU da un processo all'altro richiede il salvataggio dello stato di evoluzione della computazione del vecchio processo e il caricamento dello stato di esecuzione di quello nuovo, questo perché bisogna garantire che il processo si evolva come se fosse l'unico eseguito dal processore, quindi che nessun altro processo si permetta di toccare le informazioni che definiscono lo stato di evoluzione della computazione. Questa fase è nota come **cambio di contesto** (*context switch*), dove per contesto di un processo si intendono alcune informazioni contenute nel suo specifico PCB, come il valore dei registri, il suo stato, lo stack pointer, ecc.
Per quanto riguarda il salvataggio dello stato di evoluzione della computazione, si deve tener conto che:

- di codice del programma, dati del programma, heap e stack possiamo anche non preoccuparci, dato che il sistema operativo garantirà che non vengano toccati da nessun altro processo
- i registri vengono invece utilizzati da tutti i processi, quindi occorre salvarne da qualche parte (generalmente nello stack) il contenuto per poi recuperarli. Salvo tutti i registri o solo quelli che potrebbero essere modificati? Li salvo tutti, per tre buoni motivi:
  - per sapere quali registri saranno utilizzati dal processo successivo bisognerebbe eseguirlo, ed eseguendolo perderei tutto, non risolvendo nulla
  - anche se fosse possibile conoscere i registri che saranno modificati, ci vorrebbe troppo tempo per copiare uno alla volta quelli necessari (ogni copia corrisponde ad un ciclo *fetch-decode-execute*). Il tempo perso per il salvataggio dei registri utili va poi moltiplicato per tutte le volte che effettuo una turnazione dei processi in esecuzione, ottenendo rallentamenti spropositati
  - la maggior parte dei processori hanno un'istruzione PUSH ALL, che salva in una volta sola i valori di tutti i registri
- lo stack pointer non può essere ovviamente salvato nello stack, o otterrei un salvataggio ricorsivo (per memorizzare l'indirizzo della sua cima deve scrivere in cima, ottenendo un nuovo indirizzo della cima da memorizzare, che... ecc). E' per questo motivo che, come abbiamo visto prima, il suo valore viene memorizzato nel *process control block* associato al processo

Quando la CPU dovrà riattivare un dato processo, cercherà il suo stack pointer nel PCB relativo e da lì recupererà i valori dei registri e del program counter, così da poter riprendere l'esecuzione da dove l'aveva interrotta.
Per alcuni processi questo sistema può essere critico, a causa dei tempi di gestione delle turnazioni, la cui velocità varia da macchina a macchina (dipende da vari fattori, quali la velocità della memoria, il numero di registri da copiare, ...). I processi in questione sono proprio quelli non performanti per il multi-tasking, ovvero quelli che gestiscono sistemi in tempo reale o che comunque non gradiscono il time sharing.

Sospensione e riattivazione dei processi

Sospendere e riattivare i processi è l'anima del multi-tasking, il cui obiettivo - ricordiamolo - è consentirne la turnazione sul processore massimizzandone lo sfruttamento. Ciò viene realizzato secondo una precisa metodologia:

- *sospensione del processo in esecuzione*, salvando in modo sicuro lo stato di evoluzione della computazione in modo tale da poter tornare ad eseguire il processo dallo stesso punto in cui l'avevo lasciato
- *ordinamento dei processi in stato di pronto* (scheduling), per stabilire quale deve essere eseguito per primo
- *selezione del processo in stato di pronto da mettere in esecuzione* (dispatching), il passo logico successivo allo scheduling
- *riattivazione del processo selezionato*

Tali operazioni devono tener conto del comportamento dei processi rispetto all'uso delle risorse fisiche, in base al quale li distinguiamo in:

- ***I/O-bound***, che effettuano più I/O che computazioni
- ***CPU-bound***, che effettuano principalmente computazioni (in casi limite potrebbero anche non avere alcuna I/O)

Il sistema che implementerà il multi-tasking dovrà dunque tener conto delle classi dei processi, bilanciandoli opportunamente. Ad esempio, eseguire pochi processi I/O-bound non sfrutterebbe al meglio il processore, dandogli poco e niente da computare; eseguire troppi CPU-bound invece, pur assicurando un uso intensivo della CPU, rallenterà drasticamente il sistema (il processo monopolizza il processore e tutti gli altri non potranno evolvere). Quindi non basta massimizzare lo sfruttamento della CPU, occorre anche che l'evoluzione dei programmi in esecuzioni appaia fluida e parallela all'utente.

Questo/a opera è pubblicato sotto una [Licenza Creative Commons](#).

Come realizzare allora il sistema? Dobbiamo distinguere tra **politiche** ("*che cosa fare*") e **meccanismi** ("*come sarà fatto*"). Le prime sono le regole, che sono indipendenti da come verranno poi implementate. Nel nostro caso definiscono quando un processo debba essere sospeso, e con quale criterio si dovrà ordinare la coda dei processi pronti. I meccanismi sono invece l'insieme delle operazioni che realizzeranno la politica.
Più in dettaglio, la politica di sospensione dei processi stabilisce che un processo in esecuzione può essere sospeso in modo *implicito*, ovvero ad opera dell'ambiente operativo, o *esplicito*, in cui è il processo stesso che richiama volontariamente l'operazione di rilascio del processore.
Appartengono alla sospensione implicita i seguenti casi:

- quando viene effettuata una richiesta di I/O. E' tipica di tutti i sistemi operativi, e tiene conto dei tempi di attesa legati ai tempi elettro-meccanici di esecuzione della chiamata I/O richiesta
- quando dopo aver creato un sottoprocesso ne attendo la terminazione

Ultima considerazione è che sia la sospensione implicita che quella esplicita sono sincrone rispetto la computazione, con la differenza che la prima avviene in stato supervisor (essendo legata a chiamate di sistema, sarà il sistema operativo a gestirla), mentre la seconda in modalità utente (magari attivata con una trap). Si possono avere anche casi di sospensione asincrona rispetto la computazione, ovvero ad esempio allo scadere del quanto di tempo nei sistemi time sharing, un concetto che affronteremo nel capitolo successivo.
Vediamo ora i meccanismi. La sospensione del processo in esecuzione avviene in due passaggi: l'attivazione della procedura di sospensione (tra quelle elencate prima) ed il salvataggio del contesto di esecuzione, ovvero tutti i registri del processore nello stack e lo stack pointer nel Process Control Block. La riattivazione del processo avviene intuitivamente seguendo la procedura inversa, con il ripristino del contesto di esecuzione.
Il *context switching* visto nel capitolo precedente è dunque composto da queste due "macro-operazioni" di sospensione del processo in esecuzione e riattivazione del processo da mettere in esecuzione.
Da notare infine che alcuni processori (ad esempio gli x86) possono farlo anche in hardware (c'è un'istruzione apposita che salva il contesto dove indicato), ma a volte farlo in software risulta più veloce.

Time Sharing

Una buona definizione di sistema **time sharing** è *sistema multi-tasking a condivisione di tempo*. Ha come obiettivo quello di gestire la turnazione dei processi sul processore in modo da creare l'illusione di evoluzione contemporanea delle computazioni, come se ogni processo avesse tutta la CPU per sé. Fin qui nulla di nuovo rispetto al comune multi-tasking; la differenza sta nell'introduzione del concetto di **quanto di tempo** e **pre-emption**. Il *quanto di tempo* (*time slice*) è l'intervallo di tempo massimo di uso consecutivo del processore consentito a ciascun processo, così che si abbia una equa ripartizione della CPU; la *pre-emption* è invece il rilascio forzato del processore, con la sospensione del processo in esecuzione. Ho quindi più processi da eseguire caricati in memoria, la cui turnazione è soggetta alle stesse politiche viste per il multi-tasking con l'aggiunta della possibilità di sospensione asincrona alla computazione, dovuta allo scadere del quanto di tempo.
Il dispositivo che rende possibile la sospensione di un processo ancora in esecuzione allo scadere del time slice, è il **Real-time clock** (*RTC*). Esso non è altro che un chip impiantato sulla scheda madre contenente un cristallo di quarzo che viene fatto oscillare in modo estremamente stabile con segnali elettrici. Tali oscillazioni (i cui valori dovrebbero essere riportati sul manuale del calcolatore, e in alcuni casi sono anche programmabili) scandiscono il tempo generando periodicamente delle interruzioni da inviare al sistema operativo.
Il periodo dell'RTC è molto piccolo, e se usassi direttamente il suo valore come time slice il sistema perderebbe più tempo a gestire gli interrupt che ad eseguire i processi (questo fenomeno prende il nome di *sovraccarico di gestione dell'interruzione*). La soluzione è moltiplicarlo per un opportuno fattore *k*, così che solo un'interruzione ogni *k* verrà considerata come termine del quanto di tempo. Il fattore dovrà tener conto di un aspetto importante degli interrupt, cioè che quando il processore ne gestisce uno disabilita la ricezione degli altri. Quindi potrebbe succedere che durante la gestione dell'interruzione di altre periferiche arrivi il segnale RTC ed il processore non se ne accorga. Basterebbe però mantenere il valore dell'intervallo di tempo un po' più basso di quanto si vorrebbe, così da aumentare la frequenza degli interrupt e ridurre statisticamente l'incidenza del problema.

Accenni di schedulazione

Se con la multiprogrammazione è possibile avere più processi caricati in memoria, è grazie al multi-tasking e alla gestione time-sharing che essi possono alternarsi nell'utilizzo della CPU, così che la loro evoluzione possa essere portata avanti in parallelo.

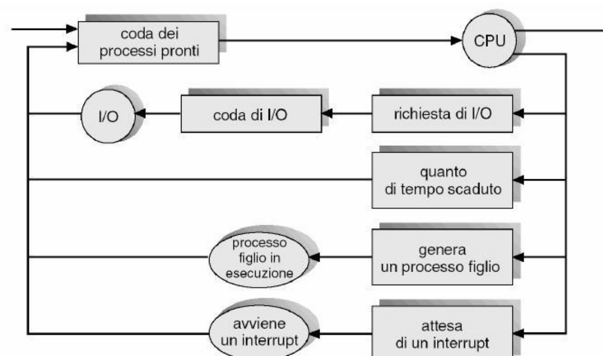
http://www.swappa.it

Questo/a opera è pubblicato sotto una Licenza Creative Commons.

Se ho un unico processore, il sistema non potrà mai avere più di un processo in esecuzione, ed ho quindi bisogno di qualcosa che selezioni i processi pronti all'esecuzione per eseguirli effettivamente sulla CPU, lasciando i rimanenti in attesa. Questo componente prende il nome di **schedulatore dei processi** (o *scheduler*), e le strutture da cui acquisisce informazioni sui processi e i lori stati si chiamano *code di schedulazione*.
Tra le code di schedulazione ricordiamo:

- la *coda di lavori* (*job queue*), in cui vengono inseriti tutti i processi del sistema al momento della loro creazione
- la *coda dei processi pronti* (*ready queue*), in cui risiedono i processi caricati in memoria centrale che si trovano nello stato *ready-to-run*. Vi usciranno solo quando saranno selezionati dallo scheduler per l'esecuzione (operazione di *dispatching*)
- la *coda delle periferiche di I/O* (*device queue*), dove vengono inseriti i processi in attesa di una particolare periferica di I/O

Tali code vengono rappresentate con il diagramma delle code, in cui ciascun rettangolo rappresenta una coda, i cerchi le risorse che li servono e le frecce il flusso dei processi nel sistema.



Si può osservare come un processo venga inizialmente messo nella coda dei processi pronti, finché non viene selezionato per l'esecuzione. A questo punto può terminare, o effettuare una richiesta di I/O e spostarsi nella relativa coda, o creare un nuovo processo e attendere che termini, o attendere un interrupt. Quando viene terminato viene rimosso da tutte le code e il suo PCB e le sue risorse vengono deallocate.

Creazione dei processi

Se durante la sua esecuzione un processo volesse crearne uno nuovo, può farlo attraverso una chiamata di sistema per la sua creazione e attivazione. Il processo generante prende il nome di *padre* e quello generato *figlio*, che a sua volta può generare nuovi figli andando a formare un *albero* di processi.
Ci sono due considerazioni da fare. La prima è che in seguito alla creazione esistono due possibilità per l'esecuzione dei processi: il padre continua la sua computazione in modo concorrente ai figli, oppure attende che tutti o parte di essi siano terminati. La seconda considerazione riguarda invece la concessione delle risorse ai figli, ovvero decidere se condividerle (tutte o in parte) col padre, fare in modo che siano ottenute direttamente dal sistema operativo, o passare dei dati in input che la inizializzino dopo la creazione.
Prima di vedere come implementare tale funzione nei sistemi Unix, anticipiamo brevemente il concetto di **spazio di indirizzamento**, cioè quella porzione di memoria centrale riservata al processo dal sistema operativo, alla quale nessun altro processo può accedere (salvo eventuali sincronizzazioni, che per ora non interessano). E' al suo interno che vengono memorizzati il codice e i dati, questi ultimi residenti nelle strutture dati opportune che abbiamo visto nei capitoli precedenti.
Possiamo finalmente vedere la chiamata di sistema per la creazione di un nuovo processo nei sistemi Unix: **fork()**. Essa produce una copia *distinta* dello spazio di indirizzamento del padre, ovvero stesso codice del programma e stessi dati *al momento della creazione*. Il meccanismo di comunicazione tra padre e figlio è particolarmente semplice, ed è reso possibile dal codice di ritorno della funzione fork(), che per il padre è l'identificatore del figlio (il *pid*, un numero intero maggiore di 0), mentre per quest'ultimo è proprio 0. Di seguito un esempio di codice:

```
int valore = fork();

if (valore == 0)
{
```

Questo/a opera è pubblicato sotto una Licenza Creative Commons.

```
  printf("Sono il figlio!\n");
  return (0);
}
else
{
  printf("Sono il padre!\n");
  return (0);
}
```

Ma se il processo figlio è una copia esatta del padre (più esattamente, è un duplicato distinto del suo spazio di indirizzamento), come si può allora generare processi indipendenti? Unix mette a disposizione un'ulteriore chiamata di sistema, la **exec()**, che permette di caricare un nuovo programma nello spazio di memoria del processo che la esegue. Il padre dovrà in pratica lanciare all'interno del processo figlio (quindi, dopo la fork()) una exec()specificando da quale porzione di memoria andare a caricare il nuovo codice e i nuovi dati da sovrascrivere agli attuali.
Pur essendo piuttosto comune che a una fork() segua una exec(), le due funzioni vengono mantenute comunque separate perché potrebbero essere utilizzate singolarmente per altri scopi diversi dalla creazione dei processi. Ad esempio con una exec() potrei cambiare il codice dello stesso processo in esecuzione semplicemente per svolgere attività di tipo diverso.

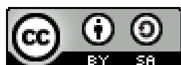
Terminazione dei processi

Un processo normalmente termina dopo l'esecuzione della sua ultima istruzione, quando chiede al sistema operativo di rimuoverlo tramite la chiamata **exit()**. Per rimozione dal sistema si intende la deallocazione di tutte le risorse in uso, ovvero le porzioni di memoria occupate, i file aperti e i buffer di I/O. La exit() restituisce un valore di ritorno (tipicamente un intero) dal quale si capisce se l'operazione è andata a buon fine o se ho riscontrato errori.
Esistono altri due motivi per i quali può terminare un processo, entrambi che coinvolgono il padre dello stesso. In un caso ho la cosiddetta *terminazione diretta*, col padre che decide di interrompere l'esecuzione di un suo figlio per vari motivi (perché ha ecceduto nell'uso delle risorse concesse, perché i suoi servizi non sono più necessari, ...) attraverso la funzione abort(). L'altro prende invece il nome di *terminazione a cascata*, è messa in esecuzione da alcuni sistemi operativi e consiste nella terminazione forzata di tutti i figli di un processo che (in modo normale o anormale) ha finito la sua computazione. Altri sistemi invece, come ad esempio l'Unix, nel caso in cui un processo termini, assegnerebbero tutti i suoi figli al processo *init* (il primo processo che il kernel manda in esecuzione dopo aver terminato il bootstrap) così che possano continuare la loro evoluzione.

Thread

Dal processo tradizionale...

I processi tradizionali sono programmi in esecuzione con un unico flusso di controllo. Ognuno di essi è un'entità autonoma, ciascuna con il proprio spazio di indirizzamento inaccessibile agli altri. Ma se questa proprietà da un lato garantisce integrità e sicurezza, dall'altro può rappresentare un problema, soprattutto per quelle applicazioni ad alta disponibilità di servizio e basso tempo di risposta (come i server web), dove l'utente - o gli utenti - potrebbero richiedere l'esecuzione di più flussi di controllo nello stesso processo per attività simili.
Ad esempio ho un server web che accetta le richieste concorrenti di più client per le pagine web e tutti gli oggetti in esse contenute. Se fosse eseguito come un singolo processo riuscirebbe a servire un solo client per volta, facendo lievitare i tempi di attesa di tutti gli altri.
Una soluzione semplice potrebbe essere attivare tanti processi che erogano il servizio quante sono le richieste per lo stesso (ovviamente stiamo parlando di sistemi multiprogrammati e multitasking), ma anche in questo caso avrei svantaggi piuttosto rilevanti. In primo luogo sarebbe impossibile prevedere a priori il numero di richieste di accesso, e quindi quanti processi dovrò attivare. In secondo luogo avrei un sistema lento, con una gestione poco furba delle risorse. Se infatti al server web arrivassero più richieste per una stessa pagina, ogni volta il processo di servizio dovrebbe leggerla e inviarla al client, sprecando tempo e spazio.
Sarebbe dunque bello se operazioni simili su dati diversi operassero in modo più correlato, magari su un'area di memoria centrale condivisa a cui i processi possano accedere in modo nativo e naturale. In particolare, vorrei che le applicazioni rendessero disponibili i propri dati ai processi figli in modo semplice, senza doverli ogni volta ricopiare nei rispettivi spazi di indirizzamento, perché è una soluzione inefficiente.



Questo/a opera è pubblicato sotto una Licenza Creative Commons.

...al concetto di thread

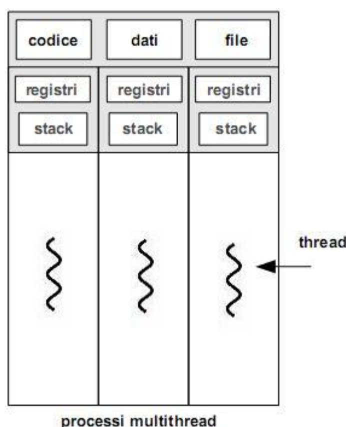
E' sulla base di queste necessità che nascono i **thread**, ovvero *un gruppo di flussi di esecuzione autonomi sullo stesso programma che accedono alla stessa porzione di memoria centrale*. Detto in altre parole, è un mini-processo che vive all'interno di un processo vero, autonomo dal punto di vista dell'esecuzione, ma con la stessa "base di dati" a cui attingere, ovvero quella del processo che lo ha generato.

Il thread può essere anche considerato come l'unità base dell'utilizzo della CPU, e comprende un identificatore, un program counter, un set di registri e uno stack, e condivide con gli altri thread che appartengono allo stesso processo la sezione di codice, quella dei dati e altre risorse del sistema operativo (ad esempio i file aperti).

Processi multithread

Se il processo tradizionale (detto *pesante*) ha un solo thread che fa tutto, un processo *multithread* è caratterizzato da più flussi di esecuzione di istruzioni in parallelo, operanti contemporaneamente e con parte delle informazioni condivise in memoria centrale. In questo modo ogni thread svolgerà le proprie operazioni e potrà eventualmente trasmetterne i risultati a flussi diversi della computazione, proprio in virtù della condivisione dell'accesso allo stesso spazio di indirizzamento. Ovviamente andranno sincronizzati opportunamente, o potrei avere dati inconsistenti (perché ad esempio modificati da altri).

Un processo multithread è rappresentato nella figura sottostante.



Si può notare come, pur avendo in comune lo stesso codice, due thread diversi hanno comunque contesti specifici su cui operare (quindi registri e stack separati), in modo da poter eseguire operazioni diverse prese da parti diverse del codice. Se infatti condividessero anche stack e registri, finirebbero inevitabilmente per eseguire le stesse operazioni nello stesso momento.

Benefici

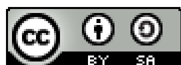
I benefici della programmazione multithread sono i seguenti:

- *prontezza di risposta*, dato che l'eventuale disponibilità di thread liberi garantisce una maggiore rapidità nel fornire il servizio all'utente. Ad esempio, un browser multithread può permettere all'utente l'interazione in un thread mentre con un altro carica un'immagine;
- *condivisione nativa delle risorse*, dei thread tra loro e con il processo a cui essi appartengono;
- *economia nell'occupazione delle risorse*, risparmiando memoria nella rappresentazione delle informazioni nel sistema. La creazione e la gestione dei thread è più economica di quella dei processi anche in termini di tempo, ad esempio in Solaris la creazione è 30 volte più veloce;
- *sfruttabili dai sistemi multiprocessore*, dove i thread possono essere eseguiti in parallelo su processori differenti, raggiungendo così una condivisione efficiente e diretta delle informazioni.

Supporto

Il supporto per i thread può essere realizzato a due livelli:

- nello *spazio utente*, per gli *user thread*. Sono supportati al di sopra del kernel e sono gestiti senza che lui ne sappia nulla: è il processo che gestisce i suoi thread interni, il sistema operativo sa solo che esiste, ma non sa cosa fa. Tale supporto è reso possibile dalle librerie di thread, residenti completamente nello spazio utente, che forniscono al programmatore le API per la loro creazione e gestione. Invocare una funzione di libreria si traduce quindi in una chiamata di funzione locale nello spazio utente e non in una chiamata di sistema.

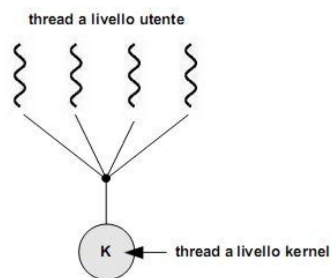

Questo/a opera è pubblicato sotto una Licenza Creative Commons.

- nel *kernel*, per i *kernel thread*. Sono supportati e gestiti direttamente dal sistema operativo, implementando ad esempio una libreria a livello kernel. Al contrario di prima, in questo caso il codice e le strutture dati della libreria risiedono nello spazio kernel, e quindi l'invocazione a una sua funzione si traduce in una chiamata di sistema.

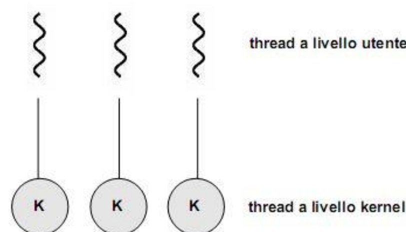
Modelli multithread

Ora che abbiamo distinto thread a livello utente e a livello kernel dobbiamo chiederci come possono essere messi in relazione tra loro, ovvero che *modelli multithread* sono realizzabili.

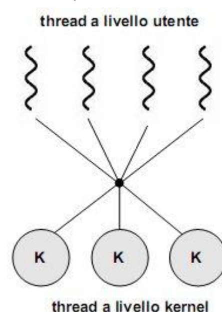
- *modello molti-a-uno*, che riunisce molti thread di livello utente in un unico flusso di controllo sul kernel. La gestione dei thread è fatta dalla libreria nello spazio utente, quindi sarà il programma a dover sincronizzare opportunamente i suoi thread interni in modo che il kernel thread che li gestisce ne veda uno solo. Lo svantaggio di tale mappatura è che se uno dei thread in esecuzione si blocca (ad esempio per un I/O), verranno bloccati di conseguenza tutti gli altri thread che fanno parte del suo gruppo. Per fare un esempio, il thread kernel A gestisce i thread utente Alfa, Beta e Gamma. Se Beta fa un'I/O, si blocca in attesa del risultato. Bloccandosi Beta, anche A si blocca perché è proprio A a dover fornire l'I/O a Beta. Ma se A smazza l'I/O di Beta, allora Alfa e Gamma non possono fare altro che dormire, perché A, che dovrebbe gestirli, non può ascoltarli in quanto impegnato a dare retta all'I/O di Beta! Basta dunque una chiamata bloccante perché tutto il sistema si arresti



- *modello uno-a-uno*, che mappa ciascun thread utente in un kernel thread. Fornisce molta più concorrenza del modello precedente, permettendo ad un thread di essere eseguito nonostante un'eventuale chiamata bloccante da parte di un altro. Consente inoltre su sistemi multiprocessore che più thread siano eseguiti in parallelo su diversi processori. Lo svantaggio di tale modello è che la creazione di un thread utente richiede la creazione del corrispondente kernel thread, con un conseguente overhead di gestione che grava sul sistema operativo diminuendone proporzionalmente l'efficienza complessiva.

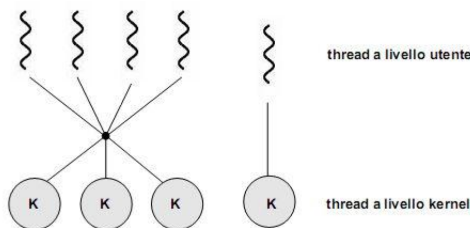


- *modello molti-a-molti*, che raggruppa in vario modo i thread a livello utente verso un numero inferiore (o equivalente) di kernel thread. Tale numero può dipendere sia dall'applicazione che dalla macchina utilizzata. Con questo modello vengono superati entrambi i limiti dei due precedenti, perché non ho più i problemi di concorrenza del *molti-a-uno* né limiti nella creazione di thread utente come nell' *uno-a-uno*. Col *molti-a-molti* posso infatti creare quanti thread voglio, che potranno comunque essere eseguiti in parallelo.



Questo/a opera è pubblicato sotto una Licenza Creative Commons.

Una variante comune è quella del *modello a due livelli*, che mappa molti thread di livello utente verso un numero più piccolo o equivalente di kernel thread, ma permette anche di associarne alcuni in modalità *uno-a-uno*.



Va da sé che questi modelli si applicano su sistemi multithread. In un sistema operativo con supporto per soli processi pesanti, occorre simularli a livello utente all'interno di un processo utilizzando una libreria di livello utente.

Cooperazione tra thread

La cooperazione tra più thread può essere rappresentata secondo tre modelli di comportamento:

- *thread simmetrici*, in cui tutti i thread sono equipotenti, ovvero posso scegliere di attivarne uno qualunque per servire una richiesta esterna. Ad esempio, se in un server web mi arriva la richiesta di un client, utilizzerò uno dei thread di servizio disponibili per ascoltare la richiesta, caricare la pagina dal disco e reinviargliela.
- *thread gerarchici*, con la divisione in due livelli tra coordinatori e lavoratori (worker thread). I primi coordinano i lavori (chi fa cosa, come e quando), ricevendo le richieste esterne e decidendo eventualmente a quale thread lavoratore indirizzarle; i secondi eseguono. Notare come questo modella permetta di avere il coordinatore praticamente sempre reperibile, dal momento che evade rapidamente le sue occupazioni. A livello implementativo è conveniente mappare il thread coordinatore nel kernel, e i worker nel livello utente.
- *thread in pipeline*, dove ogni thread svolge una porzione del lavoro complessivo, essendo specializzato in un preciso sottoinsieme delle funzioni dell'elaborazione complessiva. Avviene dunque una suddivisione del lavoro, come in una catena di montaggio. Ottengo così una distribuzione dei lavori ed un throughput elevato, dal momento che ogni thread torna in attesa di soddisfare nuove richieste dopo il tempo minimale che impiega per svolgere la sua piccola sequenza di operazioni (il concetto è fare poche operazioni, ma spesso).

Gestione dei thread

Vedremo ora le varie funzioni messe a disposizione per la gestione dei thread.

Creazione

La sintassi della chiamata di sistema per creare un nuovo thread è uguale a quella dei processi, ovvero *fork()*. La semantica cambia invece sensibilmente. Il problema è il seguente: se il thread di un programma chiama una fork(), verrà creato un nuovo processo con la copia di tutti i thread, o un processo pesante composto da quell'unico thread che aveva lanciato la fork? Dipende dal sistema operativo. Ad esempio nei sistemi Unix viene data la possibilità di scegliere tra le due alternative fornendo due funzioni fork() distinte: quella che duplica tutti i thread e quella che duplica solo il thread che effettua la chiamata di sistema.

Esecuzione

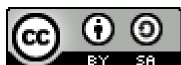
La funzione di esecuzione *exec()* riveste il thread di un nuovo codice, rimpiazzando quello di partenza. In questo caso oltre la sintassi rimane invariata anche la semantica rispetto alla corrispondente chiamata di sistema per i processi.
A questo punto diventa più semplice decidere quale tipo di fork() andare ad eseguire:

- se dovrò chiamare una exec() subito dopo la fork(), allora quest'ultima potrà essere benissimo quella che duplica il solo thread chiamante, dato che la prima operazione che eseguirà sarà cancellare sé stesso e caricare qualcos'altro
- se non viene chiamata alcuna exec() dopo la fork(), allora sarebbe più opportuno duplicare tutti i thread del processo padre.

Quindi, perché due thread appartenenti a uno stesso processo possano eseguire operazioni diverse, posso adottare due strategie: faccio delle *call()* a porzioni diverse del codice condiviso, oppure utilizzo le *exec()*.

Cancellazione

Cancellare un thread significa terminarlo prima che abbia completato la sua esecuzione, ad esempio in un browser web quando clicco sul pulsante di interruzione del caricamento della pagina.
Il thread che sta per essere cancellato viene spesso chiamato *thread target*, e la sua cancellazione può avvenire in due modalità:


Questo/a opera è pubblicato sotto una Licenza Creative Commons.

- *asincrona*, con terminazione immediata indipendentemente dall'operazione che sta svolgendo in quel momento;
- *differita*, inserendo il thread in una specie di "lista nera" e verificando periodicamente se sono in punti del codice in cui è possibile terminarlo. Altrimenti attendo semplicemente che il thread finisca la sua computazione, ottenendo di fatto una terminazione ordinaria.

Sincronizzazione e comunicazione

Se ho due processi che lavorano insieme ed ho un thread che vuole comunicare con l'altro processo, la comunicazione può avvenire con tutti i thread del processo destinatario, con un loro sottoinsieme o con uno specifico.

Processi leggeri

Abbiamo visto come nel modello multithread molti-a-molti (così come in quello a due livelli, di cui è una variante) i thread a livello utente vengano mappati su un numero inferiore (o al più uguale) di kernel thread. Diventa dunque opportuna una coordinazione tra kernel e libreria dei thread, effettuando una loro schedulazione in modo da aggiustarne dinamicamente il numero nel kernel, migliorando così le prestazioni complessive del sistema.
Se il kernel supporta nativamente tale schedulazione non ci sono problemi, ma in caso contrario? Una soluzione potrebbe essere "mascherare" i thread da processi, così che il sistema operativo possa sfruttare gli algoritmi che già conosce e implementa per ottenere il multi-tasking. E' a questo scopo che in molti sistemi sono state introdotte delle strutture dati intermedie fra i thread a livello utente e quelli a livello kernel, detti *processi leggeri* (*lightweight process, LWP*). Dei processi essi detengono tutte quelle caratteristiche che lo rendono autonomo e passibile di context-switching, così da potervi applicare la schedulazione; non ha però un proprio spazio di indirizzamento, condividendo di fatto gran parte della memoria con gli altri thread (livello utente). Sull'altro livello, l'LWP appare alla libreria dei thread utente come una sorta di *processore virtuale* grazie al quale l'applicazione può schedulare l'esecuzione di un thread utente. Va infine ricordato che la mappatura "thread utente - processo leggero" segue il modello uno-a-uno.

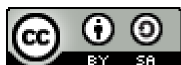
Schedulazione

Introduzione

La **schedulazione** è un'operazione fondamentale dei moderni sistemi operativi multiprogrammati e multi-tasking. La sua funzione è quella di gestire in modo ottimale la turnazione dei processi sulla CPU, definendo delle politiche di ordinamento.
Il suo successo si basa sulla proprietà di ciclicità del processo, che durante la sua esecuzione alterna continuamente fasi di computazione svolte dal processore (*picco di CPU*) e fasi di attesa di una periferica (*picco di I/O*), che si concludono con una richiesta di terminazione al sistema. La durata dei picchi del processore variano molto di processo in processo e di macchina in macchina, ma si attesta in media sotto gli 8 millisecondi. In particolare, processi di classe I/O-bound avranno molti picchi brevi della CPU, mentre quelli CPU-bound (con intensa attività computazionale) ne avranno pochi ma lunghi. Questo genere di considerazioni devono essere messe in gran rilievo nella scelta dell'algoritmo di schedulazione.
Esistono due strategie di attivazione della schedulazione, con e senza rilascio forzato.
La schedulazione senza rilascio anticipato è chiamata **non pre-emptive**: una volta assegnata la CPU ad un processo, questo la tiene finché non avvengono una delle seguenti circostanze:

- quando un processo in esecuzione passa allo stato di attesa (ad esempio, dopo aver chiesto un'operazione di I/O o attendendo il termine di uno dei figli)
- quando un processo in esecuzione passa allo stato di pronto (ad esempio, quando si verifica un interrupt)
- quando un processo rilascia volontariamente il processore
- quando un processo termina

Quando avviene uno di questi casi, viene mandato in esecuzione il processo selezionato dallo schedulatore. La schedulazione *non-preemptive* è quindi evidentemente sincrona con l'evoluzione dello stato della computazione.
Nei sistemi multi-tasking a condivisione di tempo (time sharing) si rende invece necessaria un'ulteriore modalità di attivazione dello schedulatore, asincrona con la computazione, che corrisponde allo scadere del quanto di tempo concesso al processo in esecuzione. Si parla dunque di schedulazione con rilascio anticipato, o **pre-emptive scheduling**. Essa non è del tutto indolore, dal momento che può comportare un costo legato all'accesso dei dati condivisi. Ad esempio, si considerino due processi che condividono dei dati: mentre uno li sta modificando, gli scade il


Questo/a opera è pubblicato sotto una Licenza Creative Commons.

time slice e viene terminato forzatamente. Quando l'altro proverà a leggere i dati li troverà in stato inconsistente. Bisognerà dunque trovare dei meccanismi di sincronizzazione per evitare che ciò accada.

Livelli di schedulazione

I sistemi operativi possono schedulare secondo tre modalità, detti *livelli di schedulazione*, che agiscono appunto su diversi livelli temporali. I nomi stessi sottolineano la loro dipendenza dalla frequenza con cui sono eseguiti, quindi dal fattore tempo; si parla infatti di schedulazione *a breve termine*, *a lungo termine* e *a medio termine*.

La **schedulazione a breve termine** (anche nota come *CPU-scheduler*), ha come obiettivo ordinare i processi già caricati in memoria centrale e nello stato ready-to-run. Perché proprio questi? Perché se considerassi anche i processi non attivi dovrei prima caricarli dalle memorie di massa, perdendo del tempo, così come per quelli in stato di wait, che se anche ottenessero l'accesso al processore dovrebbero comunque aspettare il verificarsi dell'evento per il quale sono in attesa.

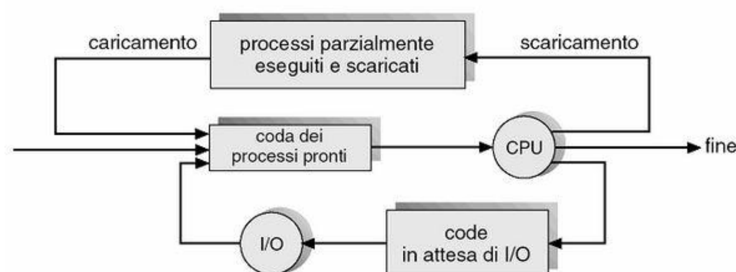
La schedulazione a breve termine garantisce quindi una turnazione rapida, ed è per questo che viene eseguita frequentemente (almeno ogni 100 millisecondi). L'algoritmo che l'implementa deve quindi essere molto veloce per minimizzare il carico di gestione, o finirei a metterci più tempo a schedulare che ad eseguire; ciò si ottiene con una relativa semplicità del codice, che opera una scelta migliore nell'immediato piuttosto che fare valutazioni più complete ma dispendiose.

La **schedulazione a lungo termine** (anche detto *long-term scheduler* o *job scheduler*) ordina tutti i processi attivati nel sistema, compresi quelli presenti nelle memorie di massa, identificando il gruppo di processi che devono essere caricati in memoria centrale per l'esecuzione e posti nello stato di ready-to-run. Tale gruppo è costruito mescolando processi CPU-bound e I/O-bound in modo da massimizzare lo sfruttamento della CPU. Ci vuole quindi equilibrio: esagerare con i CPU-bound significherebbe sfruttare al meglio il processore, ma annullare le interazioni con l'utente; utilizzare troppi I/O-bound garantirebbe invece una corretta e fluida interazione, ma con tempi di elaborazione eterni. In alcuni sistemi tale livello di schedulazione è assente o minimale. Ad esempio in un sistema embedded ho sempre tutti i processi caricati in memoria centrale, quindi un *job scheduler* si renderebbe del tutto inutile. Spesso non lo hanno neanche Unix e Windows, che si affidano quasi esclusivamente al *CPU-scheduler*, in quanto non è semplice conoscere a priori il numero di processi attivati.

Gli algoritmi sono usualmente lenti e complessi (senza esagerare), perché scegliere la combinazione più ragionevole di processi da caricare dalle memorie ausiliarie per sfruttare al meglio il processore non è affatto un lavoro banale. Deve perciò essere eseguito poco frequentemente (tipicamente una volta ogni qualche minuto) per evitare di sovraccaricare il sistema, o non sarebbe più così produttivo.

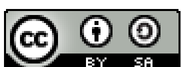
La **schedulazione a medio termine** (anche nota come *mid-term scheduler*) si pone a un livello intermedio tra i due precedenti. Viene tipicamente implementato in sistemi operativi come quello con gestione a time sharing, e fa fronte a diversi problemi prestazionali: alta concorrenza per l'utilizzo della CPU, sbilanciamento tra processi I/O-bound e CPU-bound, scarsità o esaurimento della memoria centrale disponibile. Il *mid-term scheduler* si propone di risolvere tutte queste criticità modificando dinamicamente il gruppo di processi caricati in memoria centrale e posti nello stato ready-to-run dallo *schedulatore a lungo termine* (riducendo così il grado di multiprogrammazione), al fine di adattarli all'effettiva distribuzione dei lavori che stanno compiendo. In altre parole, corregge gli ordinamenti della schedulazione a lungo termine (che non poteva prevedere tutti gli scenari a priori) in base alla situazione attuale. I processi rimossi dalla memoria centrale vengono spostati in un'area di memoria di massa temporanea, con una procedura chiamata *swapping out*; il procedimento inverso prende invece il nome di *swapping in*.

Ecco uno schema che rappresenta tale tipo di schedulazione.



Criteri e metodi di valutazione

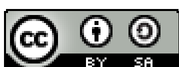
Prima di decidere quale tipo di algoritmo di schedulazione della CPU utilizzare, bisogna considerare che ognuno di essi ha caratteristiche che possono favorire una classe di processi piuttosto che un'altra. Esistono diversi criteri per la


Questo/a opera è pubblicato sotto una Licenza Creative Commons.

comparazione degli algoritmi di schedulazione, la cui valutazione preventiva può condurre alla scelta di quello migliore da utilizzare. Di seguito, alcuni criteri:

- *utilizzo del processore*, ovvero quanto riesco a massimizzarne lo sfruttamento
- *frequenza di completamento* o *throughput*, cioè quanti processi riesco a completare nell'unità di tempo (o, per vederla in un altro modo, quanti utenti riesco a soddisfare)
- *tempo di completamento* (*turnaround time*), che è l'intervallo di tempo che intercorre tra l'immissione del processo nel sistema e il suo completamento. Non considera solo il tempo di esecuzione, ma anche quello di attesa prima di essere caricato in memoria, quello speso nella coda dei processi pronti e di I/O
- *tempo d'attesa* del processo nella coda dei processi pronti
- *tempo di risposta*, più adatto del *turnaround time* nei sistemi interattivi, che indica l'intervallo di tempo che intercorre tra la formulazione della prima richiesta alla produzione della prima risposta.

In linea teorica è sempre desiderabile massimizzare l'utilizzo della CPU e il throughput, minimizzando quanto possibile il tempo di completamento, di attesa e di risposta; notare come alcuni studiosi suggeriscano di riferirsi più alle varianze di tali parametri che ai loro valori medi.
In pratica bisogna valutare attentamente il carico applicativo e i fattori critici del mio sistema, e in base a questo scegliere un algoritmo di schedulazione che sia un giusto compromesso tra i vari criteri sopra esposti. Ad esempio un server può aver bisogno di quanti di tempo lunghi, perché non deve essere interattivo ma avere tanto throughput; un sistema desktop può invece permettersi di essere meno performante sul lungo periodo, ma deve reagire istantaneamente alle richieste dell'utente.
Per vagliare se gli algoritmi considerati soddisfino i particolari criteri che voglio raggiungere, posso utilizzare diversi metodi di valutazione.
Una delle principali è la **valutazione analitica**, di cui è un esempio la *modellazione deterministica* dell'algoritmo, che ne definisce in modo matematico le prestazioni in base a un carico di lavoro predeterminato. Da un punto di vista logico è semplice, veloce e molto precisa; per contro ha però che richiede numeri esatti in ingresso, senza nessun approssimazione, perché piccoli errori sui dati potrebbero ripercuotersi pesantemente nella valutazione delle cifre di riferimento del problema analitico. Ulteriore limite è che i suoi risultati si applicano solo ai casi studiati, non possono essere generalizzati. Proprio per questi motivi la *modellazione deterministica* viene usata principalmente per la descrizione degli algoritmi di schedulazione e la produzione di esempi.
Se non è possibile creare un modello deterministico, la **valutazione statistica** mi permette di considerare da un punto di vista probabilistico i valori dei termini di riferimento che mi interessano, quindi con un certo grando di incertezza intrinseca dei dati pur sempre rappresentativi. Un esempio di questo tipo di valutazione sono i *modelli a reti di code*, in cui il sistema è rappresentato come una macchina che offre un certo numero di servizi diversi (ad esempio l'accesso alla memoria, a un disco, a una periferica, ...). Ogni servizio ha una sua coda di processi e ogni processo, nella sua evoluzione, potrà passare da una coda all'altra; la gestione ottimale dei flussi di richieste sarà obiettivo del sistema, mentre chi compie la valutazione statistica dovrà tener traccia dei vari parametri che caratterizzano le code (frequenza di arrivo e di uscita, lunghezza media, tempi medi di attesa, ...) per ottenere una stima approssimata e verosimile del modello implementato.
Un tipo di valutazione statistica più accurata degli algoritmi di schedulazione è la *simulazione*, ovvero la realizzazione software del modello del sistema, processi applicativi e schedulatore inclusi, con un certo grado di aderenza alla realtà. Il simulatore consente di modificare le variabili del sistema, tra cui il tempo e il numero e il tipo di processi coinvolti, e monitorarne le prestazioni così da compilare delle statistiche. Pur ottenendo valutazioni piuttosto accurate, le simulazioni possono essere costose perché dovrò perdere tempo a progettarle, codificarle ed eseguirle (più sono precise e più diventano onerose).
Se nemmeno il grado di accuratezza della *simulazione* mi può bastare, allora non mi resta che l' **implementazione** come modalità di valutazione. Essa consiste nella realizzazione effettiva del sistema hardware e software, così da poter fugare ogni dubbio sull'incertezza dei risultati. In questo modo è possibile effettuare una scelta dell'algoritmo di scheduling in base alle esigenze e alle caratteristiche reali, raccogliendo le informazioni sul carico di lavoro utilizzando strumenti di rilevazione interni al sistema operativo. Anche questa soluzione comporta però alcuni svantaggi. Uno ovvio è il costo elevato, non solo in termini di sforzi di implementazione dell'algoritmo (e alle modifiche del sistema operativo perché possa supportarlo), ma anche perché comporta la cooperazione degli utenti, a molti dei quali non interessa che lo schedulatore sia più efficiente, ma che i loro processi siano eseguiti correttamente e in tempo utile.
Un'altra difficoltà è legata al fatto che il sistema gestisce carichi diversi a seconda del momento della giornata, e quindi



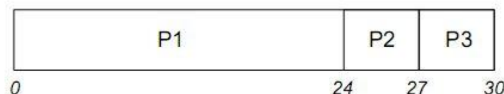
Questo/a opera è pubblicato sotto una Licenza Creative Commons.

potrebbe aver bisogno di scheduling diversi per ottenere il miglior risultato possibile in ogni momento. Si dovrà dunque valutare se usare il trend medio come criterio di valutazione, oppure adattarsi di volta in volta alla nuova situazione.

Algoritmi di schedulazione

First-Come, First-Served

L'algoritmo di schedulazione più semplice è il **First-Come, First Served** (*FCFS*), la cui politica è definita nel nome stesso: il processo che chiede la CPU per primo la ottiene per primo. L'implementazione è piuttosto immediata, potendo sfruttare strutture dati come le code *FIFO*, in particolare la coda dei processi pronti. Quando un processo passa allo stato di ready-to-run, il suo PCB viene collegato alla base della coda; quando invece il processore si libera, il processore in testa alla coda viene rimosso da essa e messo in esecuzione. Se il vantaggio di questa realizzazione è la semplicità di scrittura e lettura, uno dei limiti è rappresentato dal tempo medio di attesa piuttosto lungo. Facciamo un esempio. Ho tre processi: P1 (tempo di elaborazione, in millisecondi: 24), P2 (3) e P3 (3), che arrivano in quest'ordine. Posso visualizzare la situazione con il seguente *diagramma di Gantt*:



Il tempo di attesa è 0 per P1, 24 per P2 e 27 per P3. Il tempo medio di attesa è pari quindi a 17 millisecondi.
Se invece avessi avuto la seguente disposizione dei processi

i tempi di attesa sarebbero stati 0 per P2, 3 per P3 e 6 per P1, con un'attesa media drasticamente ridotta a 3 ms. Quello che abbiamo appena visto è un esempio di *effetto convoglio*, caratterizzato dal fatto che l'evoluzione di più processi brevi è ritardata dall'attesa per la terminazione di un unico processo più lento. Calando il tutto in un situazione reale, potremmo pensare ai processi brevi come I/O-bound, e a quello più lungo come un CPU-bound.

Shortest-Job-First

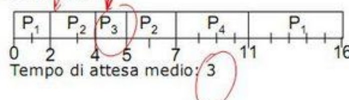
Anche nel caso del **Shortest-Job-First** (*SJF*) il nome rende l'idea che sta dietro all'algoritmo, ovvero che il processo più breve viene eseguito per primo. Tale politica di schedulazione è quella ottimale, poiché fornisce sempre il minor tempo di attesa medio per un dato gruppo di processi. Se infatti eseguo prima i processi più brevi il loro tempo di attesa diminuisce molto più di quanto possa aumentare quello dei processi più lunghi, diminuendo di conseguenza il tempo medio.

Posso implementare l' *SJF* sia pre-emptive che non pre-emptive. Come sappiamo, nel primo caso appena un processo diventa pronto interrompe quello in esecuzione richiedendo una schedulazione; nulla di tutto questo nel secondo caso. In generale si dimostra più efficiente il pre-emptive, come si può osservare nell'esempio seguente

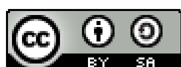
Processo	P_1	P_2	P_3	P_4
Tempo di arrivo	0	2	4	5
Tempo di elaborazione	7	4	1	4

Non pre-emptive

Pre-emptive



La complessità dell'algoritmo sta nel conoscere il tempo di elaborazione di tutti i processi, e non è un'informazione semplice da reperire. Negli schedulatori a lungo termine è possibile utilizzare come valore il tempo limite specificato dall'utente nel momento della richiesta di esecuzione del processo, ottenendo buoni risultati. Ciò non è invece possibile in quella a breve termine, dove non esiste modo di conoscere con certezza il tempo di elaborazione richiesto dai processi. Tuttavia esistono delle tecniche per prevederlo, ad esempio ipotizzando che rimanga simile a quelli precedenti si può supporre che il suo valore sia pari alla loro media esponenziale. Rimane comunque una predizione, con tutte le incertezze che essa comporta; bisogna dunque prestare molta attenzione alle scelte dei valori, o potrei fare più danni che altro.



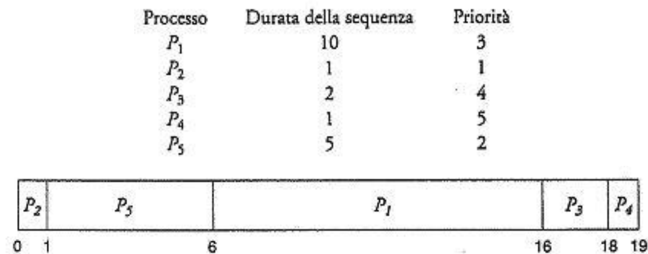
Questo/a opera è pubblicato sotto una Licenza Creative Commons.

Schedulazione a priorità

L'algoritmo di **schedulazione a priorità** assegna una priorità ad ogni processo, dove per priorità si intende una proprietà che indica quanto prima dobbiamo eseguire un processo rispetto ad altri, indipendentemente dall'ordine con cui è arrivato. La *SJF* non ne è che un caso particolare, in cui la priorità era l'inverso del tempo di elaborazione, ovvero era tanto maggiore quanto minore era il tempo.

Il valore della priorità è espresso generalmente in numeri, memorizzati in un indice associato al processo. E' importante sottolineare come non sia affatto automatico il collegamento "valore alto" -> "priorità alta". Ad esempio nei sistemi Unix più l'indice è basso e più il processo è importante; in particolare i processi di sistema hanno priorità negativa (il minimo è -20) mentre quelli delle applicazioni positiva. Questo tipo di rappresentazione è detto a *logica inversa*, mentre quella *diretta* è dei casi in cui a indice alto corrisponde priorità alta.

Ad esempio:



Processo	Durata della sequenza	Priorità
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Il tempo di attesa medio è di 8.2 millisecondi.

La priorità può essere definita *internamente* o *esternamente*. Internamente, quando come indici vengono utilizzate proprietà misurabili del processo, come il tempo di esecuzione, le risorse impiegate, ecc. Esternamente quando invece sono assegnate con politiche estranee al sistema operativo, come l'importanza del processo o degli utenti che l'utilizzano, o altro ancora.

Come al solito possiamo avere:

- *schedulazione pre-emptive*. Quando un processo entra nello stato ready-to-run andrà a controllare se il processo in esecuzione ha priorità minore, nel qual caso si sostituirà a lui con una pre-emption. Se invece è running un processo di priorità maggiore, l'ultimo arrivato verrà ordinato secondo i normali criteri di schedulazione per priorità
- *schedulazione non pre-emptive*, in cui anche se il processo che diventa pronto ha priorità maggiore di quello in esecuzione, non interrompe nulla. Verrà eseguito solo quando l'altro terminerà, e solo se non sono sopraggiunti altri processi ready-to-run di priorità maggiore

Un problema di questo tipo di politica di schedulazione è che processi a bassa priorità potrebbero rimanere ready-to-run per un tempo indefinito; basta infatti che continuino ad arrivare processi a priorità maggiore che gli passino avanti. Questo fenomeno prende il nome di **starvation**, o *blocco indefinito*. La soluzione è introdurre un progressivo invecchiamento delle priorità, detto **aging**. In pratica, se abbiamo un processo che rimane in starvation per un tempo abbastanza lungo, cominciamo ad aumentargli la priorità proporzionalmente al tempo di attesa, sperando che ne acquisisca (e prima o poi succederà) abbastanza priorità da essere eseguito. Una volta ottenuto il suo turno di computazione, il suo livello di priorità viene riabbassato allo stato iniziale, e riprende il ciclo di starvation -> invecchiamento -> computazione. Questa tecnica prevede dunque l'utilizzo di priorità di tipo dinamico, che aumentano con l'invecchiare del processo.

Round-Robin

L'algoritmo di **schedulazione Round-Robin** (*RR*) può considerarsi un adattamento del *FCFS* per i sistemi a condivisione di tempo; funziona infatti come un First-Come, First-Served con l'aggiunta della pre-emption per alternare i processi allo scadere di un *quanto di tempo*. Lo schedulatore tratta la coda dei processi pronti in modo circolare (quando esaurisce il giro, ricomincia daccapo), assegnando la CPU a ciascun processo per un intervallo di tempo della durata massima del time slice.

A livello implementativo la coda dei processi viene realizzata come una coda *FIFO*, in cui i nuovi processi vengono aggiunti in fondo. Quando lo schedulatore manda in esecuzione un processo imposta un timer per generare un interrupt dopo il quanto di tempo. A questo punto si possono avere due scenari: o il processo riesce a rilasciare la CPU prima del tempo concessogli, o il timer scatterà e il sistema operativo farà in modo che avvenga un context switch del processo in esecuzione, mettendolo in fondo alla coda dei processi pronti. In entrambi i casi lo schedulatore

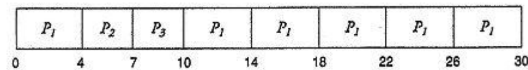
http://www.swappa.it

Questo/a opera è pubblicato sotto una Licenza Creative Commons.

selezionerà il processo ready-to-run successivo. Naturalmente l'algoritmo usato è pre-emptive, dal momento che nessun processo può essere eseguito per più di un quanto di tempo successivo (a meno che non sia l'unico pronto).
Un esempio:

Processo	Durata della sequenza
P_1	24
P_2	3
P_3	3

Se il time slice dura 4 millisecondi, ho la seguente situazione



con un tempo di attesa medio pari a 5.66 millisecondi.
Le prestazioni del *Round-Robin* non sono entusiasmanti, e dipendono molto dalla durata del time slice. Se questo è troppo grande si otterrebbe di fatto un *FCFS*, con tutti i limiti che ne derivano; se al contrario è troppo breve si ottiene la cosiddetta *condivisione del processore*, che dà ad ognuno degli *n* processi del sistema l'impressione di essere l'unico eseguito su un processore con *1/n* capacità computazionale. Quest'ultima situazione comporta però un sovraccarico della gestione del sistema, dovuto ai frequenti cambiamenti di contesto.
Bisogna dunque riporre molta attenzione nella scelta del valore del quanto di tempo, non dovendo essere né troppo alto né troppo basso. A tal fine esiste una regola empirica che raccomanda di fare in modo che l'80% delle richieste di elaborazione riesca ad essere completata in un quanto di tempo. Generalmente tale valore si attesta tra i 10 e i 100 millisecondi.
Per concludere, un'ultima considerazione sulla velocità di esecuzione dei processi e sul turnaround. I primi sono strettamente collegati al numero di processi pronti, mentre i secondi dipendono fortemente dalla durata del time slice.

Coda a più livelli

Se sono presenti nel sistema molti processi di tipo diverso si può provare a raggrupparli per tipologie omogenee, così da applicare ad ognuna di esse l'algoritmo di schedulazione più performante. Ad esempio, una classificazione comune distingue i processi eseguiti in *foreground* (quelli più interattivi, magari schedulati con un *RR*) da quelli in *background*(che agiscono sullo sfondo, per i quali si potrebbe utilizzare un *FCFS*). L'algoritmo di **schedulazione con coda a più livelli** (*C+L*) permette tale gestione, partizionando le code dei processi pronti in più code di attesa separate, ognuna col suo specifico algoritmo di schedulazione. I processi sono assegnati in modo permanente a una sola coda, scelta in base a qualche proprietà intrinseca come la dimensione di memoria, la priorità o altro.
Le code sono a loro volta schedulate da un algoritmo dedicato, usualmente di tipo pre-emptive a priorità fissa. Ad esempio potrei progettare un algoritmo di schedulazione con coda a più livelli, con quattro code, che suddivide i processi nelle seguenti tipologie:

1. di sistema
2. interattivi
3. batch
4. degli studenti

Ogni coda ha priorità maggiore di quella sottostante, e viceversa. Ciò significa che non potrà essere eseguito nessun processo nella coda dei batch se prima non si siano svuotate le code dei processi di sistema e di quelli interattivi.
Le code potrebbero schedulate anche per partizionamento di tempo, ad esempio assegnando una certa quantità di tempo di CPU a ognuna di esse. Riprendendo il primo esempio, potrei assegnare l'80% del tempo ai processi foreground, e il rimanente 20% a quelli in background.

Coda a più livelli con retroazione

La **schedulazione con coda a più livelli con retroazione** (*C+LR*) permette ciò che la *C+L* non consentiva, ovvero che un processo possa muoversi tra le code di schedulazione. Ciò rende la struttura più flessibile, adattandola all'uso dinamico del processore da parte dei processi. Se ad esempio un processo utilizza per troppo tempo la CPU, c'è il rischio di starvation per tutti i processi delle code a priorità più bassa; si attua dunque una *politica di degradazione*, ovvero lo si sposta nella coda di schedulazione di livello inferiore. Analogamente, se un processo attende troppo a lungo che sia eseguito, verrà attuata una *politica di promozione* che lo sposterà in una coda a priorità più alta. Questo schema tende a spostare i processi interattivi e I/O bound nelle code superiori, in una sorta di processo di invecchiamento dinamico che previene il fenomeno della *starvation*.
Ricapitolando, per definire lo schedulatore con coda a più livelli con feedback, oltre a specificare il numero di code e il

Questo/a opera è pubblicato sotto una Licenza Creative Commons.

tipo di algoritmo di schedulazione applicato su ognuna di essa, è necessario sapere quando implementare la politica di promozione e degradazione dei processi, e quale *politica di allocazione* attuare (ovvero, in quale coda dovrà entrare un processo quando avrà bisogno di un servizio).
Il *C+LR*, pur essendo il più completo e configurabile tra i vari algoritmi di schedulazione, è anche il più complesso di tutti.

Schedulazione per sistemi multiprocessore

La schedulazione è un'operazione strettamente vincolata al sistema di elaborazione in uso, e quando viene applicata a sistemi multi-processore la sua progettazione va affrontata in modo diverso, più complesso. Bisogna anzitutto considerare le caratteristiche dell'architettura del sistema:

- se i processori sono equipollenti (identici dal punto di vista funzionale) vengono detti *omogenei*; se ognuno di essi è invece specializzato per una funzione diversa, si parla di processori *eterogenei*. Nella nostra trattazione ci concentreremo sui primi
- la memoria può essere condivisa o meno
- le periferiche possono essere accessibili indistintamente da tutti i processori, o solo da alcuni

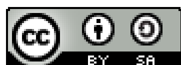
Questi fattori condizionano fortemente la scelta dell'algoritmo di schedulazione, vediamo in che modo.
In sistemi con processori omogenei e memoria e periferiche condivise possiamo attuare una politica di **suddivisione del carico** (*load sharing*), fornendo una coda separata per ciascun processore. In questo modo potrebbe però succedere che la coda di un processore rimanga vuota e quindi questo non computi; si previene il fenomeno utilizzando un'unica coda dei processi pronti per tutte le CPU, che vengono schedulati sulla prima che si libera. Da considerare però che tutti i processori leggono dalla stessa memoria, e quindi potrei subire tracolli sul suo bus a causa del sovraccaricamento.
Il *load sharing* è possibile anche se i processori hanno memoria locale oltre a quella condivisa. Ciò è possibile perché spostare un processo da un processore all'altro comporta anche lo spostamento del suo spazio di indirizzamento. In questo caso devo però tener conto del costo (tempo e spazio) di trasferimento, che nel caso precedente non c'era.
Se ho processori che condividono tutto tranne le periferiche (non per come sono realizzati loro, ma per l'assenza di un bus che li colleghi), va creata una coda relativa alla periferica a cui solo quella CPU ha accesso. Se poi oltre alle periferiche anche le memorie sono locali, allora si avranno code per ogni entità non condivisa, con gli immaginabili problemi di gestione che ne conseguono.
Un caso particolare è quello dei processori eterogenei, in cui esistono code per ogni processore e, sopra di esse, una coda per ogni gruppo di processori omogenei.
Concludendo, indipendentemente da omogeneità o eterogeneità dei processori, il *multiprocessamento* può essere di due tipi:

- *asimmetrico*, in cui il sistema operativo (quindi tutte le decisioni di schedulazione, di attività di I/O e le altre attività di sistema) sono eseguite da un singolo processore, definito *master*, e tutti gli altri pensano solo a eseguire i processi applicativi
- *simmetrico*, dove ogni processore esegue il sistema operativo e i processi applicativi, che in questo caso saranno schedulati da lui stesso.

Schedulazione per sistemi in tempo reale

I **sistemi in tempo reale** (*real-time*) sono quelli in cui il tempo è un fattore critico, ovvero dove la correttezza del risultato della computazione dipende dal tempo di risposta. Per descrivere le politiche di schedulazione di tali sistemi bisogna distinguere tra *sistemi in tempo reale stretto* e *in tempo reale lasco*.
Nei **sistemi in tempo reale stretto** (*hard real-time*) deve essere garantito il completamento di un'operazione critica entro un certo intervallo di tempo. Per vederla sotto un'altra angolazione, bisogna fare in modo che un processo termini la sua computazione entro un tempo massimo garantito dalla sua attivazione.
Esistono diverse tecniche di scheduling:

- quando un processo viene avviato, gli viene associata un'informazione sulla quantità di tempo entro il quale deve essere terminato o deve eseguire un I/O. Lo schedulatore dovrà fare una stima del tempo di elaborazione, e solo se riuscirà a garantire la terminazione del processo lo accetterà e gli prenoterà le risorse necessarie. Questa garanzia si chiama **resource reservation** e presuppone che lo schedulatore conosca con precisione i tempi di esecuzione di ogni funzione del sistema operativo, il che non è banale (basta che ci siano memorie secondarie o virtuali perché tali stime diventino imprecise).



Questo/a opera è pubblicato sotto una Licenza Creative Commons.

- se i processi sono periodici, posso associare ad ognuno di essi tre informazioni: il tempo fisso di elaborazione *t* una volta ottenuta la CPU, una scadenza *d* entro cui deve essere servito e il suo periodo *p*. La relazione tra queste grandezze è: $0 <= t <= d <= p$. Posso sfruttare queste proprietà facendo sì che il processo dichiari allo schedulatore la frequenza *1/p* e la propria scadenza, imponendo che uno dei due valori sia considerato come priorità. L'algoritmo di schedulazione applica poi una tecnica di *controllo dell'ammissione*, che verifica la possibilità di completamento entro la scadenza dichiarata con la politica di schedulazione descritta.
- anche lo **scheduling a frequenza monotòna** si applica ai processi periodici, ed è un'estensione della politica precedente con l'aggiunta di pre-emption. La priorità viene stabilita in modo inversamente proporzionale al periodo, quindi è maggiore nei processi più frequenti, e si assume che il tempo di elaborazione di un dato processo Pi rimanga uguale ad ogni accesso alla CPU. Ad esempio, ho due processi P1 (periodo: 50 ms ; tempo elab: 20 ms) e P2 (100 ms ; 35 ms). Essendo più frequente, P1 viene eseguito per primo e completa la sua elaborazione a 20 ms. Subentra P2, che però non riesce a completare la sua: dopo 30 ms (quindi tempo complessivo: 20 + 30 = 50 ms , ovvero il periodo di P1) l'altro processo a priorità maggiore lo scalza e computa per altri 20 ms. Nel suo turno, P2 dovrà essere eseguito per quei 5 ms che gli mancavano, terminando dopo 75 ms dall'inizio, e quindi stando dentro il suo periodo (di 100 ms).
  L'algoritmo è considerato ottimo perché se non riesce a schedulare lui, non riuscirebbe a farlo nessun altro algoritmo con priorità statiche in ambiente hard real-time. Ciononostante, l'utilizzo del processore è piuttosto limitato.
- l'algoritmo di **schedulazione a scadenza più urgente** (*Earliest-Deadline First*, *EDF*) si può applicare sia su processi periodici che non periodici, e anche su quelli con tempo di elaborazione variabile. Come politica assegna dinamicamente le priorità a seconda delle scadenze dei processi (che devono dichiararle): prima è la scadenza, più alta è la priorità. Ho quindi una proporzionalità inversa, *1/d*. Si dice che l'assegnamento è dinamico, perché quando un processo diventa ready-to-run le priorità di tutti gli altri processi nel sistema possono essere modificate per riflettere la scadenza del nuovo rispetto a quelli già presenti.

Nei **sistemi in tempo reale lasco** (*soft real-time*) la computazione è meno restrittiva, dal momento che richiede solo che i processi critici ricevano maggiore priorità di quelli meno importanti. Implementare tali funzionalità richiede in primo luogo operare una distinzione tra processi critici e non, stabilendo che i primi siano quelli real-time e che abbiano priorità statica (il che si ottiene facilmente impedendo l'aging). I secondi sono tutti gli altri processi, su cui è eventualmente possibile applicare una schedulazione a priorità dinamica per evitare fenomeni di starvation. Si rende poi necessario ridurre quanto possibile la *latenza del dispatch*, ovvero il tempo che intercorre tra lo stato ready-to-run e il running. Un sistema potrebbe ad esempio introdurre dei *pre-emption point* nelle chiamate di sistema particolarmente lunghe, ovvero dei punti di sospensione in cui viene controllato se un processo a priorità più alta deve essere eseguito. Dato però che metterne molti è poco pratico (c'è da considerare che vanno messi in punti sicuri), la latenza di dispatch può essere comunque lunga anche quando vengono utilizzati. Un'altra tecnica più radicale è rendere tutto il kernel interrompibile, proteggendolo per quanto possibile con meccanismi di sincronizzazione.
Ultima considerazione va fatta sul problema dell' *inversione di priorità*, ovvero quando un processo che deve leggere o modificare dati del kernel che sono in quel momento utilizzati da altri a più bassa priorità, deve attendere che questi abbiano finito. Ciò è facilmente risolvibile introducendo il **protocollo di ereditarietà**, in base al quale viene stabilito che tutti i processi che stanno accedendo alle risorse necessarie per un processo a più alta priorità, ereditano momentaneamente tale priorità finché non terminano la computazione.

Schedulazione dei thread

I concetti e le problematiche affrontate finora sono applicabili sia alla schedulazione dei processi che a quella dei thread; in particolare, per questi ultimi vanno fatte alcune considerazioni. Pur essendo mappati tra loro, esiste una separazione netta tra thread a livello utente e quelli a livello kernel (indipendentemente dall'utilizzo di processi leggeri). Ciò comporta due *livelli di schedulazione*:

- a *livello di processo*, in cui il processo farà uso di uno schedulatore presente nella libreria dei thread a livello utente. Si parla di *process-contention scope* (*PCS*, visibilità della competizione tra processi), dal momento che la competizione per il processore avviene tra thread dello stesso processo. In linea di principio può essere usata qualsiasi politica di schedulazione, anche se è preferibile usare quelle tipiche come il *FCFS*, il *RR* o a priorità.

Questo/a opera è pubblicato sotto una Licenza Creative Commons.

- a *livello di sistema*, per selezionare quale kernel thread debba effettivamente ottenere l'accesso alla CPU. In questo caso di parla di *system-contention scope* (*SCS*, , visibilità della competizione sul sistema) ed è gestita dallo schedulatore del sistema operativo.

Comunicazione tra Processi

Processi cooperanti

Un processo che non condivide dati con altri processi è detto **indipendente**, e la sua evoluzione non può influenzare o essere influenzata da alcuno. Da notare che ciò non implica che non abbiano risorse condivise (sarebbe impensabile, dal momento che almeno il processore deve essere accessibile a tutti), l'importante è che nessuno acceda alle informazioni dell'altro. I processi **cooperanti** sono invece l'esatto contrario e, pur svolgendo ognuno un compito ben definito, le loro computazioni concorrono all'adempimento di uno scopo applicativo congiunto. Stessa distinzione può essere operata sui thread.

Alcuni dei vantaggi della cooperazione sono:

- *condivisione delle informazioni*, poiché molti utenti potrebbero essere interessati a stessi dati e/o risorse
- *parallelizzazione*, che sfrutta in modo nativo l'eventuale disponibilità di più processori, velocizzando la computazione
- *modularità*, che consente di progettare l'applicazione un aspetto per volta, per poi mettere insieme in modo opportuno i moduli che implementano tali aspetti
- *scalabilità*, che permette di ampliare le capacità operative dell'applicazione aggiungendo semplicemente nuove istanze dello stesso processo (ad esempio utile per un server web)
- *specializzazione*, grazie alla quale è possibile assegnare la realizzazione dei processi che si occupano dei vari aspetti (ad esempio controllo, acquisizione, interfaccia, ecc) ad esperti specializzati in quell'ambito di programmazione
- *qualità del progetto e della realizzazione*, perché sviluppando ogni aspetto in modo separato posso raggiungere qualità elevata.

Perché tutto ciò sia possibile occorre che tali processi possano scambiarsi informazioni e coordinarsi, o meglio, che possano **comunicare** e **sincronizzarsi**. I processi indipendenti si limitano invece alla sola sincronizzazione, per regolamentare l'accesso alle risorse condivise.

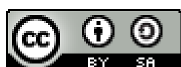
Comunicazione

Il processo di comunicazione comporta la descrizione di politiche e meccanismi che permettano ai processi di scambiarsi informazioni per operare in modo cooperativo. Anzitutto vanno definite le entità coinvolte, quindi il processo mittente (P) produttore dell'informazione, il processo ricevente (Q) utilizzatore della stessa, e il canale di comunicazione (mono o bidirezionale) attraverso cui farla fluire. Per decidere poi quale tipo di comunicazione è meglio adottare in una specifica situazione, bisognerà tener conto di alcune caratteristiche. Di seguito riportiamo le più importanti, alcune delle quali dovrebbero sempre essere garantite:

- *quantità di informazioni da trasmettere*. Se avessi ad esempio grosse quantità di informazioni, potrebbe essere poco conveniente copiarle in memoria centrale (sprecherei molto tempo)
- *velocità di esecuzione*
- *scalabilità*, descritta prima. Bisognerebbe infatti garantire in modo semplice la possibilità di aumentare i processi che concorrono alla comunicazione, cercando di non far lievitare la complessità totale
- *semplicità di uso nelle applicazioni*
- *omogeneità delle comunicazioni*, dato che non avrebbe senso adottare *n* modi diversi di effettuare una comunicazione, col conseguente incremento delle complessità di gestione e quindi di commettere errori
- *integrazione nel linguaggio di programmazione*, che garantisce la portabilità. E' sempre consigliabile utilizzare le tecniche di comunicazione definite nel linguaggio di programmazione utilizzato, o sarei costretto ad usare librerie che potrebbero cambiare di sistema in sistema
- *affidabilità*
- *sicurezza*
- *protezione*

Prima di elencarne le principali implementazioni, distinguiamo le due modalità di realizzazione:

- **Comunicazione diretta**, in cui ogni processo che voglia comunicare deve conoscere esplicitamente il nome del destinatario o del mittente della comunicazione. Notare come tra ogni coppia di processi possa sussistere



Questo/a opera è pubblicato sotto una Licenza Creative Commons.

un'unica connessione, e come si presupponga che P e Q siano entrambi vivi e attivi per tutta la durata della comunicazione (il che rappresenta spesso un grosso limite)

- **Comunicazione indiretta**, in cui mittente e ricevente non si conoscono e la comunicazione avviene su punti noti a entrambi (i canali) sfruttando una struttura dati passiva per contenere l'informazione. Se il problema della modalità diretta era che se una delle entità in gioco veniva a mancare durante la comunicazione nessun altro poteva accedere all'informazione, nella modalità indiretta (essendo il canale in comune) ci sarebbe sempre qualche processo in grado di raccogliere l'informazione.

Implementazioni

Con memoria condivisa

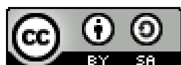
E' una modalità di comunicazione diretta e può essere realizzata attraverso l'uso di *variabili globali* o *buffer*.
Nella **condivisione di variabili globali** ho due processi con una porzione del loro spazio di indirizzamento (le variabili globali, appunto) che si sovrappone. Pur essendo un sistema molto rapido, avere porzioni di memoria in comune implica dei problemi, come la possibilità di compiere operazioni inconsistenti. Se ad esempio un processo richiede la lettura di un'informazione in corso di modifica dall'altro processo, leggerebbe qualcosa di obsoleto e dunque errato. La situazione è però facilmente risolvibile attuando meccanismi di sincronizzazione per l'accesso in mutua esclusione all'area dati comune per le operazioni non compatibili (come ad esempio lettura e scrittura contemporanea). Inoltre sono in qualche modo tutelato dal fatto che essendo i due processi cooperanti non hanno alcun interesse a danneggiare l'altro, dal momento che indirettamente danneggerebbero sé stessi. Ho due meccanismi per realizzarla.
Il primo è fare in modo che l' *area comune sia copiata dal sistema operativo*, che crea l'illusione della condivisione spostando l'informazione in due tempi: 1. la copia dall'area condivisa del processo mittente al proprio spazio di indirizzamento (in una porzione di memoria grande abbastanza) 2. da qui la copia nell'area condivisa del ricevente. Ricordiamo che il sistema operativo può penetrare ovunque, quindi non ha alcun problema ad accedere agli spazi di indirizzamento riservati dei processi. Il limite di questo meccanismo è proprio che l'operazione di copiatura si fa in due passaggi, il che su grandi porzioni di memoria rallenterebbe sensibilmente il sistema; soprattutto se si considera il fatto che anche se un processo modifica solo una minima parte dell'area condivisa, il sistema operativo non ne conosce l'entità e quindi trasferisce tutto anche se non sarebbe necessario.
Un secondo meccanismo è la *realizzazione con area comune fisicamente condivisa*, in cui il sistema operativo garantisce che l'area comune appartenga contemporaneamente agli spazi di indirizzamento di entrambi i processi. Lo spazio rimane comunque logicamente separato e garantito protetto dal sistema operativo, anche se alcune porzioni sono residenti fisicamente negli stessi indirizzi. E' piuttosto semplice da realizzare e supera tutti i limiti della soluzione precedente, bisogna solo fare in modo che non avvengano operazioni inconsistenti (basta utilizzare politiche di sincronizzazione).
Nella **condivisione di buffer** ciò che cambia rispetto a prima è l'utilizzo di un buffer. La comunicazione rimane diretta, ma il processo mittente scriverà stavolta le sue informazioni all'interno del buffer dal quale poi il processo ricevente andrà a recuperarle. Anche in questo caso con la sincronizzazione è possibile imporre gli accessi in mutua esclusione così da garantire la consistenza.
I meccanismi con cui realizzarli sono del tutto simili a quelli precedenti, ovvero *buffer con copiatura gestita dal sistema operativo* e *buffer in memoria fisicamente condivisa*.
Notare infine come i buffer siano significativamente più piccoli delle porzioni di memoria centrale condivisa, quindi le operazioni di copiatura sono molto più veloci.

Con scambio di messaggi

La **comunicazione con scambio di messaggi** è diretta e prevede che l'informazione viaggi incapsulata all'interno di messaggi. Oltre ad essa, i messaggi contengono l'identità del processo mittente e ricevente, ed eventuali altre informazioni relative alla gestione dello scambio.
I messaggi vengono memorizzati in buffer che il sistema operativo può assegnare esplicitamente ad ogni coppia di processi, o predisporne un certo numero di uso generale che mette a disposizione di chiunque ne abbia bisogno. Tale quantità può essere:

- *illimitata*: il processo mittente appena possiede un messaggio lo deposita immediatamente nel buffer. Non è mai bloccante;
- *limitata*: il processo mittente potrebbe aspettare prima di depositare il messaggio nel caso in cui non ci siano buffer liberi. Risulta quindi bloccante se manca spazio disponibile;

http://www.swappa.it



Questo/a opera è pubblicato sotto una Licenza Creative Commons.

- *nulla*: il processo mittente non può depositare alcun messaggio dato che non ci sono buffer disponibili in cui deporlo. E' sempre bloccante. Da notare che in questo scenario il processo ricevente non troverà il messaggio, ma il mittente in attesa di deporlo; ciò permetterà l'avvio di una comunicazione diretta

Le funzioni messe a disposizione per la gestione dei messaggi sono quelle per l' *invio* e la *ricezione*, eventualmente condizionali:

- **Invio**: *send(proc_ricevente, messaggio)*.
  Deposita il messaggio in un buffer libero. La funzione è bloccante, ovvero blocca il processo mittente nel caso in cui non ci fosse spazio disponibile. Una volta liberato un buffer, la funzione completa la comunicazione con il destinatario e sblocca il mittente.
- **Invio condizionale**: *cond_send(proc_ricevente, messaggio): error status*.
  A differenza della funzione di invio precedente, non è bloccante. Se al momento di depositare il messaggio non è presente alcun buffer libero, ritornerà un messaggio di errore e il messaggio non sarà più depositato. Sarà responsabilità del mittente decidere se rimandarlo o meno.
- **Ricezione**: *receive(proc_mittente, messaggio)*.
  Riceve il messaggio presente nel buffer. E' bloccante, ovvero blocca il destinatario fino a quando non c'è un messaggio nel buffer da leggere.
- **Ricezione condizionale**: *con_receive(proc_mittente, messaggio): error status*.
  Il processo ricevente preleverà il messaggio dal buffer; se non ci sono messaggi ritornerà una condizione di errore, senza bloccare il destinatario.

La comunicazione tramite buffer in generale è *asincrona*, ovvero il mittente può spedire il messaggio in qualsiasi momento della computazione senza preoccuparsi se c'è qualche ricevente in grado di raccoglierlo. In una comunicazione *sincrona* lo scambio delle informazioni avviene invece solo quando entrambi gli interlocutori sono pronti. Si può ottenere utilizzando un buffer di dimensione nulla, così che tutte le operazioni di scrittura diventino bloccanti, obbligando di fatto i processi a scrivere e leggere nello stesso momento. E' molto interessante poi osservare che tale politica può essere facilmente adottata per modalità di comunicazione diretta o indiretta. Nel primo caso si parla di *comunicazione simmetrica* e prevede ovviamente che mittente e destinatario siano sempre univocamente identificati. Nel secondo caso otteniamo invece una *comunicazione asimmetrica*, in cui l'accesso ad un buffer in scrittura non è limitato ad un processo solo, ma ad un gruppo di processi che possono a loro volta decidere di far leggere il loro messaggio a tutti quelli che ascoltano o solo ad uno di essi.

Dal momento che la sincronizzazione per l'accesso ai messaggi viene gestita implicitamente dal sistema operativo, i buffer vengono implementati direttamente all'interno del suo spazio di indirizzamento. In questo modo viene risolto anche il problema dell'identificazione dei processi che inviano informazione, dato che è sufficiente vedere qual è il processo attivo (se ha effettuato una richiesta è evidentemente nello stato di running). I messaggi vengono normalmente smaltiti in modalità *FIFO*, ma potrebbero essere previste politiche di altro tipo (ad esempio con priorità, deadline, ecc).

Comunicazione con mailbox

La **comunicazione con mailbox** è un sistema indiretto, in cui non c'è conoscenza esplicita tra i processi che comunicano. Se nella *comunicazione asimmetrica* vista in precedenza non era necessaria la conoscenza esplicita dei processi, ma era sufficiente l'identificativo dei gruppi, questo sistema è invece completamente anonimo. Il messaggio viene depositato in una **mailbox** (detta anche *porta*), una struttura dati presente nel sistema operativo caratterizzata da un nome cui si farà riferimento per accedere ai messaggi in essa contenuta. Questi contengono, oltre all'informazione da trasmettere, anche l'identità del processo mittente, il nome della mailbox destinataria ed eventuali altre informazioni di supporto alla gestione dei messaggi nella mailbox (ad esempio se vanno tolti messaggi con deadline scaduta o come ordinare i messaggi). Va ricordato che non stiamo assegnando i messaggi ad una coppia di processi, ma li stiamo inserendo in una struttura accessibile (teoricamente) da tutti; eventuali restrizioni potranno essere applicate con delle politiche di accesso stabilite dal sistema operativo.

In generale la mailbox non è proprietà del processo, ma del sistema operativo (ricordiamo che è nel suo spazio di indirizzamento). Esistono però delle procedure per attribuirle ad un processo proprietario, al quale il sistema può dare il diritto esclusivo di ricezione. In questo caso, quando il processo proprietario termina, la mailbox viene deallocata con esso.

Similmente al buffer, la mailbox può avere una capienza limitata illimitata o nulla, indipendentemente dal numero di processi che la utilizzano e con le stesse conseguenze viste prima. Anche le funzioni sono simili:

Questo/a opera è pubblicato sotto una Licenza Creative Commons.

- **Invio**: *send(M, messaggio)*
  Deposita il messaggio nella mailbox indicata da parametro.
- **Invio condizionale**: *cond_send(M, messaggio): error status*
- **Ricezione**: *receive(M, messaggio)*
- **Ricezione condizionale**: *con_receive(M, messaggio): error status*

e in più abbiamo

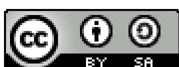
- *create(M)*
  Crea la mailbox con M nome simbolico associato
- *delete(M)*
  Cancella la mailbox indicata

Come per i buffer, le politiche di sincronizzazione delle mailbox dipendono dalla loro capacità. Se è illimitata ho una comunicazione asincrona, ovvero il mittente depone il suo messaggio indipendentemente dallo stato di computazione del processo ricevente. Se invece è nulla quella che ottengo è un comunicazione sincrona. Ho infatti un processo mittente che non trova spazio per deporre il suo messaggio, ed è dunque obbligato a mettersi in attesa. Quando un processo Q invocherà una receive(), lo troverà ancora in quello stato, e a questo punto i due processi cominceranno a passarsi l'informazione direttamente in modo sincrono. Questo meccanismo prende il nome di*rendez-vou*, ed implica che P sappia a che punto della computazione si trovi il processo ricevente quando richiede la comunicazione. Se infine la capacità è limitata, la comunicazione verrà *bufferizzata*, ovvero passerà un po' di tempo prima che il messaggio inviato venga ricevuto.
Per quanto riguarda l'ordinamento delle code dei messaggi all'interno delle mailbox, può essere applicato un qualsiasi algoritmo di schedulazione.
La *comunicazione con mailbox* è particolarmente adatta per i seguenti scenari:

- *comunicazioni molti a uno*, dove l'unico processo che può leggere dalla mailbox è detto *processo di servizio* (o*server*) e tutti i richiedenti sono i *client*. Nel caso un cui il server abortisca, il sistema operativo ne esegue immediatamente una copia diversa solo per l'identificatore. Nessun client si accorge della situazione, tranne quello eventualmente in comunicazione col processo di servizio originale quando era terminato. Il nuovo server non è infatti messo a conoscenza delle richieste pendenti
- *comunicazioni uno a molti*, con più processi di servizio che soddisfano le richieste di un unico mittente. Per quanto questo possa produrre messaggi, ci sono buone possibilità che trovi sempre un ricevente disponibile. Questa tecnica è auspicabile quando ho un singolo processo che ha più richieste che vuole siano soddisfatte contemporaneamente
- *comunicazione molti a molti*, in cui diversi processi di servizio comunicano con diversi processi client.

Comunicazione con file
Distinguiamo due implementazioni, quella con *file condivisi* e quella mediante *pipe*.
La **comunicazione mediante file condivisi** rappresenta una diretta estensione della comunicazione con le mailbox: se quella faceva uso di strutture dati realizzate in memoria centrale condivisa, i file sono invece memorizzati in memoria di massa. La sua gestione sarà al solito demandata al sistema operativo, che garantirà che le operazioni di accesso siano fatte in modo corretto e solo dai processi autorizzati.
La **comunicazione mediante pipe** impiega invece l'utilizzo dei *pipe* come strutture di appoggio, una struttura dati di tipo *FIFO* residente in memoria centrale e che condivide coi file molte delle loro funzioni.
Utilizzo per entrambi le stesse funzioni viste per le mailbox, delle quali ereditano anche gli stessi problemi (e soluzioni) di sincronizzazione e ordinamento.
Comunicazione con socket
La **comunicazione con socket** è dal punto di vista concettuale una generalizzazione in rete delle *pipe*, anche se dal punto di vista realizzativo è ovviamente qualcosa di molto più complesso. In sostanza ho il sistema operativo che virtualizza la comunicazione tra macchine diverse utilizzando una pipe spezzata in due porzioni, ognuna residente sulla memoria centrale delle macchine coinvolte.
L'architettura che sta dietro a tale modalità di comunicazione è di tipo client-server, ovvero ho un client che inoltra una richiesta ad una porta specifica su cui un server è in ascolto per la loro evasione. L'identificazione del server avviene tramite la coppia di identificatori *indirizzo:porta*.

http://www.swappa.it


Questo/a opera è pubblicato sotto una Licenza Creative Commons.

I messaggi possono avere dimensione fissa o variabile a seconda delle applicazioni che le utilizzano, e sono ordinati in modalità *FIFO* (così come i processi in attesa). I socket si possono creare o distruggere, e si può leggerci o scriverci sopra tramite due canali mono-direzionali. Ciò significa che nel momento in cui una connessione via socket viene accettata, è possibile sia per il server che per il client comunicare su due canali separati. Per evitare che i client trovino la porta di ascolto del server chiusa mentre sta servendo un altro processo, il client chiamante fa in modo di segnalare al server una nuova porta su cui spostare la comunicazione per la gestione della risposta.
Infine, la connessione può essere con gestione del sistema operativo o senza, in cui ogni messaggio viene trasmesso in modo individuale. Questa seconda modalità è ovviamente la più semplice da realizzare, dal momento che non viene fornita alcuna informazione su come il sistema dovrà trattare i messaggi. C'è poi un'ultima tecnica di connessione, che è il multicast.

Sincronizzazione tra i Processi

Concorrenza

La **concorrenza** sussiste quando ho più processi che richiedono l'accesso a risorse condivise usabili solo in mutua esclusione. Tali processi vengono definiti *concorrenti*. Sarà detto qui una volta per tutte, tutto ciò di cui si parlerà in questa sede si applica sia ai processi che ai thread, a meno di notifica.
Per *mutua esclusione* si intende banalmente che non possono essere compiute contemporaneamente sulla stessa risorsa operazioni incompatibili da parte di più processi. Ad esempio una scrittura e una lettura sullo stesso dato non sono concesse, due letture sì. Il suo scopo è dunque preservare la consistenza dell'informazione, o si avrebbero casi in cui i processi computino dati scorretti. La **sincronizzazione** è quindi quell'insieme di politiche e meccanismi che si occuperanno di garantire tale principio per l'uso di risorse condivise, che queste siano fisiche (come le periferiche) o informative (come file o altri dati). Notare come sia un problema peculiare dei sistemi multi-tasking, dal momento che se avessi un'esecuzione seriale dei processi non avrei mai richieste di accesso parallele.

Corse e sezioni critiche

Il *problema del produttore-consumatore* è un modello classico per lo studio della sincronizzazione nei sistemi operativi. Esso consiste nel sincronizzare due processi, uno (il *produttore*) che vuole inviare informazioni (sui suoi prodotti) e l'altro (il *consumatore*) che vuole leggerli. Entrambi i processi utilizzano un buffer circolare di *n* elementi come struttura dati di appoggio condivisa. I codici potrebbero essere i seguenti:

Produttore

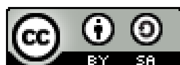
```
while (count == BUFFER_SIZE) ; // non fa nulla
// aggiungi un elemento nel buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```

Quando il produttore vorrà inviare delle informazioni, se il buffer non è pieno le inserirà nel primo spazio vuoto disponibile, incrementando di uno il valore di ''count'.

Consumatore

```
while (count == 0) ; // non fa nulla
// rimuovi un elemento nel buffer
--count;
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
```

Se il consumatore trova il buffer vuoto aspetta, altrimenti legge l'informazione e decrementa di uno il contatore degli elementi presenti nel buffer.
Il problema nasce nella condivisione della variabile *count*. Se ad esempio scade il time slice del processo produttore proprio quando sta per incrementare il contatore, avrei risultati inconsistenti dato che l'inserimento non sarà segnalato al consumatore che accederà a valori obsoleti. Queste situazioni in cui più processi accedono agli stessi dati in maniera concorrente e il risultato dell'esecuzione dipende dall'ordine in cui ha avuto luogo l'accesso, si chiamano **corse critiche**. Ricorrono molto frequentemente e non sempre conducono a effettiva inconsistenza delle informazioni. Il problema è che possono farlo, dunque vanno evitate.
Quindi, la corsa critica del problema del produttore-consumatore non è legata ad operazioni contemporanee di lettura



Questo/a opera è pubblicato sotto una Licenza Creative Commons.

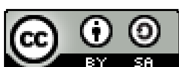
e scrittura, dal momento che ognuna di esse avviene all'interno dello spazio di indirizzamento dei rispettivi processi; ma è dovuta ai possibili errori nell'aggiornamento del contatore del buffer. Le operazioni di incremento e decremento sono infatti realizzate con una sequenza di istruzioni assembler, e proprio in quanto sequenza per definizione interrompibili.
La **sezione critica** è una porzione di codice che può generare *corse critiche* se eseguita in modo concorrente, ad esempio modifiche di variabili comuni, scrittura su file condivisi, eccetera. Lo scopo della sua individuazione è quello di creare un protocollo che possa essere utilizzato per bypassare i problemi di inconsistenza; la soluzione deve dunque soddisfare tre condizioni:

- *mutua esclusione*, ovvero se un processo sta eseguendo la sua sezione critica nessun altro può farlo
- *progresso*, che stabilisce che solo chi *non* sta eseguendo la propria sezione critica può concorrere con gli altri per accedervi, e la scelta non può durare indefinitamente
- *attesa limitata*, bisogna cioè fare in modo che nessun processo attenda troppo a lungo di evolversi

Riassumendo, le sezioni critiche di codice devono avere accesso esclusivo alle variabili condivise, e devono sfruttarle in modo rapido perché altri processi che usano la stessa risorsa non debbano attendere indefinitamente. Ciò si ottiene progettando *processi cooperanti*, che superino le criticità con un'opportuna sincronizzazione dell'evoluzione delle loro computazioni. L'idea di base è che ognuno di loro sappia a che punto della computazione si trova l'altro prima di entrare in una sezione critica, e che questi interrompa la computazione fintanto che il primo vi rimane.
Variabili di turno
Un tipo di approccio a livello di istruzioni per la sincronizzazione di due processi concorrenti, sono le **variabili di turno**. Esse sono variabili condivise tra i processi che interagiscono per accedere in modo concorrente a una risorsa, stabilendone il turno d'uso. In altre parole dicono quale processo ha il diritto di usarla in un certo istante. Vediamo tre possibili implementazioni che garantiscono la mutua esclusione tra due processi **0** e **1**.
Algoritmo 1
Riportiamo parte del codice, e in seguito i commenti.

```
...
 private volatile int turn;  // la variabile di turno
 public Algorithm 1() // (1)
 {
  turn = TURN 0;
 }
 public void enteringCriticalSection (int t) // (2)
 {
  while (turn != t)
    Thread.yield();
 }
 public void leavingCriticalSection (int t) // (3)
 {
  turn = 1 - t;
 }
...
```

(1) Inizializza a *0* la variabile di turno, dando così la possibilità al processo **0** di accedere alla propria sezione critica.
(2) Il processo **0** può accedere all'uso della risorsa condivisa, mentre a un qualsiasi altro processo (con valore *t*diverso da 0) viene invece impedito l'accesso, lasciandolo bloccato nella funzione di attesa yield(). Ovviamente, se fosse il processo **1** quello che sta utilizzando in quel momento la risorsa, sarebbe **0** quello mantenuto in attesa.
(3) Quando il processo **0** (o **1**) avrà terminato le sue operazioni nella sezione critica, concederà l'uso delle risorse condivise all'altro processo. Ciò avviene cambiando il valore alla variabile *turn*, in questo caso con l'assegnamento "1 - t" (perché ho solo due processi).
Questo algoritmo garantisce la mutua esclusione imponendo una stretta alternanza dei processi, con tutti gli svantaggi che questo comporta. Se ad esempio ho un produttore che vuole deporre più informazioni durante il suo turno, non potrà farlo perché dovrà prima aspettare che il consumatore esaurisca il suo (e viceversa). In più, non viene garantito il progresso, dato che non lo lascio evolvere fino ai suoi limiti naturali; nel caso del consumatore essi sono l'aver letto tutte le informazioni, mentre per il produttore è ritrovarsi col buffer pieno.


Questo/a opera è pubblicato sotto una Licenza Creative Commons.

Algoritmo 2
Associa un flag per ogni processo in modo che sia impostato a *true* quando sono nella sezione critica, *false* altrimenti.
Come prima, riportiamo parte del codice e in seguito i commenti.

```
...
 private volatile boolean flag0, flag1;
 public Algorithm 2() // (1)
 {
  flag0 = false; flag1 = false;
 }
 public void enteringCriticalSection (int t) // (2)
 {
  if (t == 0)
  {
   flag0 = true;
   while (flag1 == true)
     Thread.yield();
  }
  else
  {
   flag1 = true;
   while (flag0 == true)
     Thread.yield();
  }
 }
 public void leavingCriticalSection (int t) // (3)
 {
  if(t == 0) flag0 = false; else flag1 = false;
 }
...
```

(1) Inizializza entrambi i flag a *false* per l'ovvio motivo che i processi devono ancora fare richiesta di entrare nelle rispettive sezioni critiche.
(2) Se il processo che vuole accedere alla risorsa è lo **0**, flag0 passa a *true*. Nel caso in cui flag1 sia già a *true* (e quindi stia già usando la risorsa condivisa), attende prima che finisca.
(3) Quando un processo termina le sue operazioni sulla sezione critica, pone il proprio flag a *false*.
Questo algoritmo garantisce la mutua esclusione ma senza imporre una stretta alternanza dei processi, dato che chi fa richiesta di accesso a una risorsa controlla dai valori dei flag se gli è consentito o meno. Continua però a non garantire il progresso, e per di più potrebbe portare un processo a un'attesa infinita.

Algoritmo 3
Utilizza sia i flag che la variabile di turno, i primi per la prenotazione della risorsa, la seconda per stabilire chi ha accesso. Riportiamo parte del codice e in seguito i commenti.

```
...
 private volatile boolean flag0, flag1;
 private volatile int turn;
 public Algorithm 3()
 {
  flag0 = false; flag1 = false;
  turn = TURN 0;
 }
 public void enteringCriticalSection (int t) // (1)
 {
  int other = 1 - t;
  turn = other;
```

Questo/a opera è pubblicato sotto una Licenza Creative Commons.

```
  if (t == 0)
  {
   flag0 = true;
   while (flag1 == true && turn == other)
     Thread.yield();
  }
  else
  {
   flag1 = true;
   while (flag0 == true && turn == other)
     Thread.yield();
  }
 }
 public void leavingCriticalSection (int t)
 {
  if(t == 0) flag0 = false; else flag1 = false;
 }
...
```

(1) Grazie all'assegnamento turn = other;, la prossima volta che viene lanciata la funzione verrà passato other(l'altro processo) come parametro. Se quest'ultimo non deve fare nulla ripassa immediatamente il turno all'altro, altrimenti esegue le operazioni sulla risorsa di cui ha ottenuto l'accesso.
Questo algoritmo garantisce sia la mutua esclusione che il progresso, perché non rimane bloccato a causa di prenotazioni grazie alla doppia condizione del while.

Variabili di lock

La **variabile di lock** è una variabile condivisa che definisce lo stato di uso di una risorsa, cioè quando è in uso da parte di un processo (che è evidentemente nella sua sezione critica). Cambia così il punto di vista rispetto alla *variabile di turno*: non sono più i processi ad alternarsi ma è la risorsa stessa a dire se è disponibile o no. Inoltre, le variabili di turno devono essere inizializzate e gestite a seconda del numero di processi che accedono alla stessa risorsa, e nella grande maggioranza dei casi questo numero non è predicibile a priori; i *lock* invece scalano benissimo, perché non importa quanti sono i processi che vogliono una risorsa, lui sarà sempre o libero o occupato.
Può assumere due valori: *0* se la risorsa è libera, *1* se è in uso.
L'acquisizione di una risorsa avviene a interruzioni disabilitate, come si evince dai seguenti passaggi operativi:

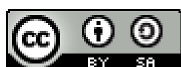
- disabilito le interruzioni
- leggo la variabile di lock
- se la risorsa è libera (*lock = 0*), la marco in uso ponendo *lock = 1* e riabilito le interruzioni
- se la risorsa è in uso (*lock = 1*), riabilito le interruzioni e pongo il processo in attesa che la risorsa si liberi
- per rilasciare la risorsa pongo *lock = 0*

Il motivo per cui vanno bloccate le interruzioni è che le operazioni di lettura ed eventuale scrittura di un *lock* sono realizzate con una sequenza di istruzioni macchina, quindi interrompibili (solo la singola istruzione macchina è atomica). Se quindi non mi tutelo sospendendo gli interrupt finirei per avere una corsa critica nello stesso sistema che uso per prevenirle.
Una soluzione alternativa è introdurre direttamente nell'hardware del processore dei meccanismi per ottenere la sincronizzazione. Ad esempio è possibile inserire l'istruzione atomica **TEST-AND-SET** che traduce la sequenza di istruzioni precedenti in una sola, non rendendo più necessaria alcuna sospensione delle interruzioni. Essa implementa infatti in un'unica istruzione le seguente operazioni: legge la variabile di lock e la pone in un flag del processore, quindi la pone uguale a uno, infine se il flag (vecchio valore di lock) era *0* allora significa che la risorsa era libera, altrimenti il processo dovrà attendere. Il limite di questa soluzione è che non spetta al programmatore la sua implementazione, bensì al progettista del processore.

Semafori

La gestione delle variabili di lock è un'operazione delicata, quindi è preferibile evitare di fare affidamento sul solo buonsenso dei programmatori, lasciando che sia il sistema operativo ad occuparsene. Ciò implica un innalzamento del livello di astrazione della sincronizzazione e una garanzia di consistenza, visto che l'esecuzione *supervisor* è a

http://www.swappa.it



Questo/a opera è pubblicato sotto una Licenza Creative Commons.

interruzioni disabilitate.
A tal fine sono stati introdotti i **semafori binari** *S*, una struttura dati (in particolare, una variabile binaria) gestita dal sistema operativo che rappresenta lo stato di uso della risorsa condivisa: ha valore *1* se la risorsa è *libera*, *0* se è *in uso*. Nonostante il significato dei valori sia opposto a quelli di lock, non c'è rischio di confondersi dato che il programmatore userà direttamente le funzioni di sistema, delegando a quest'ultimo il problema di assegnare il valore corretto.
Le funzioni succitate per la manipolazione del semaforo *S* sono atomiche proprio in quanto chiamate di sistema, e vanno utilizzate all'entrata e all'uscita della sezione critica di un processo. Esse sono:

- *acquire(S)*, che richiede l'uso della risorsa (se il valore del semaforo è *1*, lo porta a *0*)
- *release(S)*, che rilascia la risorsa (riporta il valore a *1*)

Se ho delle code di attesa posso regolarle con una qualsiasi politica di ordinamento, in tutto simili a quelle viste per la schedulazione, passando ulteriori parametri alla acquire() che diano informazioni sulla gestione della coda stessa.
Il problema dell'acquire() è che crea attese attive sul semaforo *S*, ovvero processi che pur essendo in stato di attesa continuano a effettuare computazioni per verificare se la risorsa si è liberata. Ciò comporta spreco di CPU che altri processi potrebbero usare in modo più produttivo. Questo fenomeno si chiama *spinlock* e se avviene per brevi periodi può tornare anche utile, dal momento che evita onerosi cambi di contesto (i processi rimangono sempre nello stato di *wait*). Quando però i tempi di attesa attiva diventano troppo lunghi si può adottare una tecnica diversa, ovvero spostare i processi in attesa in una coda di attesa così concepita:

- quando un processo esegue un' acquire(S), se la risorsa non è disponibile passa in stato di *wait* e viene accodato nella coda di attesa del semaforo
- quando un processo esegue una release(S), la risorsa viene rilasciata e viene attivato il primo processo della coda di attesa di *S*, a cui viene concesso l'accesso
- nel frattempo lo schedulatore dei processi in attesa della risorsa definisce l'ordine di ottenimento della risorsa stessa in base alla politica adottata

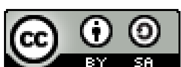
Bisogna comunque prestare attenzione alla progettazione della coda di attesa, o potrei avere fenomeni di *stallo*(*deadlock*) in cui due o più processi aspettano un evento che può essere generato solo da uno dei processi in attesa. In altre parole bisogna prestare attenzione all'ordine con cui avvengono le chiamate di sistema di prenotazione e rilascio delle risorse, o potrei avere attesa circolari senza rilascio. Un altro pericolo è quello di incorrere in una *starvation*, il blocco indefinito meglio affrontato nel capitolo sulla schedulazione.
I **semafori generalizzati** *S* sono invece variabili intere che rappresentano lo stato di uso di un insieme di risorse omogenee condivise; in altre parole, indicano il numero *n* di risorse libere del tipo richiesto. La sintassi delle funzioni di manipolazioni del semaforo rimangono le stesse, con la differenza che in questo caso l' acquire(S) (acquisizione d'uso di una risorsa) e la release(S) (rilascio della risorsa) rispettivamente incrementano e decrementano di uno il valore del semaforo. In particolare, quando l'acquire() arriva a *0* si blocca.

Monitor

Nonostante la loro semplicità ed efficacia, il problema principale della sincronizzazione con uso di semafori è che la responsabilità della loro correttezza è lasciata al programmatore. Per quanto questi possa essere preparato e attento, potrebbero comunque avvenire errori nella progettazione e/o implementazioni del codice; un esempio piuttosto grossolano potrebbe essere dimenticarsi di rilasciare una risorsa, o gestire male le code d'attesa così da avere *starvation*. E' per questo motivo che è preferibile sollevare il programmatore da questo onere, innalzando ulteriormente il livello di astrazione per la gestione della sincronizzazione lasciando che sia il sistema operativo ad occuparsene. Un costrutto fondamentale di questo tipo è il tipo di dati astratto **monitor**, che incapsula un'insieme di operazioni definite dall'utente su cui garantisce la mutua esclusione. Viene formulato a livello di linguaggio di programmazione, e al suo interno vengono definite una serie di procedure, ognuna delle quali se invocata causa l'accesso del processo a una sezione critica. La sincronizzazione è implicita, dal momento che il compilatore del linguaggio non consentirebbe l'esecuzione di più flussi concorrenti nello stesso tipo di dati.
Un esempio di pseudocodice in java è il seguente:

```
monitor nome-del-monitor
{
  // dichiarazione delle variabili

  public entry p1(..) { ... }
```



Questo/a opera è pubblicato sotto una Licenza Creative Commons.

public entry p2(..) { ... }
...
}

Perché un processo possa accedere alle risorse deve soddisfare alcune condizioni, rappresentate da variabili di tipo condition definite dal programmatore. Se il processo associato a una di esse la verifica, prosegue; altrimenti chiama una funzione speciale di wait e si mette in attesa, fino a quando un altro processo associato alle stesse condizioni lo risveglia con una signal. Notare come quest'ultima funzione non si riferisca a un processo specifico, ma a tutti quelli nella coda dei processi associati alla condizione, così che possano concorrere tra loro per avere il monitor.
Supponiamo ora di avere due processi P e Q associati a una condizione x, con Q in attesa. Se P invoca una x.signal e viene concesso al processo Q di riprendere, posso comportarmi in due modi:

- *segnala e aspetta*, in cui P attende finché Q lascia il monitor, o aspetta un'altra condizione
- *segnala e continua*, il contrario

Un compromesso tra le due è la soluzione adottata dal linguaggio Concurrent Pascal: quando il processo P esegue l'operazione signal, lascia immediatamente il monitor e Q viene subito risvegliato.

Deadlock

Introduzione

In sistemi multiprogrammati è comune che processi e thread diversi competano per l'accesso ad un numero limitato di risorse, ognuna delle quali può mettere a disposizione un certo numero di istanze identiche che soddisferanno le richieste allo stesso modo. Da questa situazione possono scaturire problemi di uso scorretto e inconsistente delle risorse, cui si fa fronte con politiche di sincronizzazione. Quindi, perché un processo possa utilizzare una risorsa dovrà effettuare le seguenti operazioni in sequenza:

- *richiesta*, che se non può essere soddisfatta immediatamente impone che il richiedente attenda il suo turno
- *uso*
- *rilascio*, passo essenziale, in cui il processo libera la risorsa

Questa soluzione non è priva di problemi, dal momento che può portare a due tipi di situazioni deleterie:
lo *starvation*e il *deadlock*. Del primo si è già parlato abbondantemente nel capitolo sulla schedulazione, ora invece tratteremo del secondo caso.

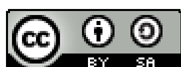
Deadlock

Un gruppo di processi si dice in uno stato di **deadlock** (o *stallo*) quando *ogni* processo del gruppo sta aspettando un evento che può essere generato soltanto da un altro processo del gruppo. Gli eventi in questione possono essere di vario tipo, nel nostro caso l'acquisizione e il rilascio delle risorse, che queste siano fisiche (componenti e periferiche) o logiche (ad esempio, file, semafori, ecc). Nella condizione di *deadlock* i processi non terminano mai la loro esecuzione e le risorse di sistema a loro assegnate rimangono bloccate, impedendone l'accesso agli altri processi.
Formalmente, ho una condizione di deadlock **se e solo se** si verificano *simultaneamente* quattro condizioni:

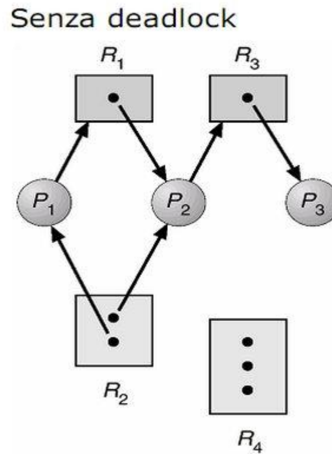
- **mutua esclusione** (*mutual exclusion*), ovvero almeno una risorsa deve essere usata in un modo non condivisibile: se due processi chiedono l'accesso a una risorsa, uno dei due deve essere messo in attesa. E' intuibile che se non avessi risorse per cui competere allora non ci sarebbe nemmeno deadlock
- **possesso e attesa** (*hold & wait*), in cui un processo che possiede già almeno una risorsa, per terminare la computazione deve attendere che se ne liberino altre occupate
- **nessun rilascio anticipato** (*no pre-emption*), perché se lo consentissi non si potrebbe mai bloccare l'accesso a una risorsa dato che chi la utilizza verrebbe forzatamente terminato
- **attesa circolare** (*circular wait*). Se ad esempio ho tre processi P1, P2 e P3, ho una situazione di attesa circolare se P1 attende il rilascio di una risorsa occupata da P2, anche lui in attesa di una risorsa associata a P3, il quale aspetta il rilascio di quella allocata a P1. Notare come questa condizione implichi anche il *possesso e attesa*

Grafo di allocazione delle risorse

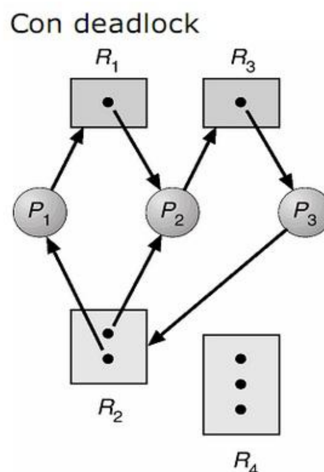
Per verificare con precisione la sussistenza di un deadlock si utilizza il **grafo di allocazione delle risorse**, ovvero un grafo costituito da un insieme di nodi che rappresentano sia i processi P_i che le risorse R_i del sistema, e di archi che si suddividono in *archi di richiesta* (che vanno da P ad R) e *archi di assegnazione* (da R a P). Se ho quindi un arco uscente da una risorsa ed entrante in un nodo, significa che quella risorsa è assegnata a quel nodo; se ho invece un arco


Questo/a opera è pubblicato sotto una Licenza Creative Commons.

uscente da un nodo ed entrante in una risorsa significa che quel nodo ha fatto richiesta di quella risorsa.
Per convenzione, i nodi P_i sono rappresentati graficamente con dei cerchi e le risorse R_i con dei rettangoli, all'interno dei quali possono esserci più puntini che rappresentano le istanze (nessun puntino -> un'unica istanza). Peculiarità degli archi è che mentre quelli di richiesta non devono indicare un'istanza in particolare, quelli di assegnazione sì. Data questa definizione, si può dimostrare che se il grafo non contiene cicli allora nessun processo del sistema è in deadlock; se invece contiene un ciclo allora ci può essere una situazione di stallo. Bisogna infatti verificare che tra le risorse coinvolte nel ciclo ci sia almeno un'istanza in più disponibile, altrimenti ho il deadlock.
Di seguito tre esempi:



Abbiamo tre processi (P1, P2, P3) e quattro risorse (R1, R2, R3, R4). R1 ed R3 hanno un'unica istanza, mentre R2 ne ha due ed R4 tre.
In particolare:
Il processo P1 ha richiesto R1 che però è già stato assegnato a P2, quindi rimane in attesa.
La risorsa R2 mette a disposizione due istanze, che pur essendo identiche possono essere usate in parallelo. Non ho conflitti dato che il principio di mutua esclusione si applica solo sulla stessa istanza.
Il processo P3 spezza la possibilità di formare un'attesa circolare, non avendo richieste pendenti su altre risorse già occupate. Una volta che ha terminato l'utilizzo di R3, lo rilascia sbloccando P2 (e per propagazione anche P1).
Non ho quindi deadlock.

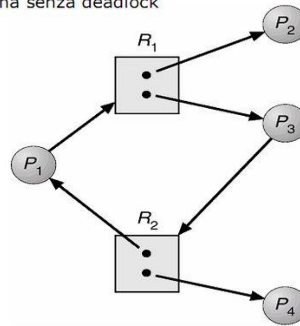


Come prima abbiamo tre processi (P1, P2, P3) e quattro risorse (R1, R2, R3, R4). R1 ed R3 hanno un'unica istanza, mentre R2 ne ha due ed R4 tre.
In particolare:
Il processo P1 ha richiesto R1 che però è già stato assegnato a P2, quindi rimane in attesa.
La risorsa R2 mette a disposizione due istanze, che pur essendo identiche possono essere usate in parallelo. Non ho conflitti dato che il principio di mutua esclusione si applica solo sulla stessa istanza.
Il processo P3 richiedendo la risorsa R2 che non ha istanze disponibili, si mette in attesa
Si sono creati così due cicli:
P1 -> R1 -> P2 -> R3 -> P3 -> R2 -> P1

http://www.swappa.it

Questo/a opera è pubblicato sotto una Licenza Creative Commons.

P2 -> R3 -> P3 -> R2 -> P2
I processi P1, P2 e P3 sono in deadlock. Il processo P2 sta infatti aspettando la risorsa R3 che è tenuta dal processo P3, il quale sta aspettando che P1 o P2 rilascino un'istanza di R2. In più il processo P1 sta aspettando che P2 liberi R1. I processi potrebbero rimanere in questo stato di attesa indefinitamente.



Abbiamo quattro processi (P1, P2, P3, P4) e due risorse (R1, R2) ognuna delle quali con due istanze.
In particolare:
Esiste un ciclo tra P1 -> R1 -> P3 -> R2 -> P1.
Ho due processi P2 e P4 che non sono coinvolti nel ciclo.
Questa situazione dimostra che se le risorse hanno più istanze, posso avere dei casi in cui ho cicli ma non deadlock. In questo caso, P2 e P4 prima o poi termineranno la loro computazione e rilasceranno un'istanza, che potrà essere utilizzata da P1 e P3 per superare lo stallo.

Metodi di gestione del deadlock

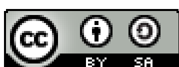
Esistono diversi metodi di gestione degli stalli, ognuno dei quali comporta un diverso carico di lavoro sul processore. Bisogna dunque scegliere quello che si rivela il miglior compromesso tra "uso di risorse di calcolo" e "criticità della presenza del deadlock" nel sistema.
Le principali strategie di gestione sono le seguenti:

- **ignorare** del tutto il problema e fingere che i deadlock non si presentino mai nel sistema. Posso permettermi di adottare tale metodo quando l'eventualità che accadano stalli sia rarissima, ad esempio legata ad un errore hardware o software che avviene con frequenza molto bassa. Questa politica normalmente consente all'utente che non vede progredire il suo processo di abortirlo o riavviare tutto il sistema. In questo modo, essendo di per sé raro il fenomeno di stallo, sarà difficile che si ripresentino nello stesso ordine le richieste alle risorse che avevano condotto al deadlock precedente. Può sembrare strano, ma questa soluzione è quella usata dalla maggior parte dei sistemi operativi, Unix e Windows compresi. Perfino la Java Virtual Machine non gestisce deadlock, viene lasciato tutto al programmatore
- **prevenzione** del deadlock (*deadlock prevention*), che garantisce *a priori*, nel momento in cui vengono fatte le richieste, che queste non generino situazioni di stallo. Consiste in un insieme di metodi atti ad accertarsi che almeno una delle condizione necessarie viste prima non possano verificarsi
- **evitare** il deadlock (*deadlock avoidance*), che non impedisce ai processi di effettuare le richieste, ma se queste potrebbero causare stalli le blocca. Richiede che i processi forniscano al sistema operativo delle informazioni supplementari su quali e quante risorse chiederà nell'evoluzione della sua computazione
- **rilevazione e recupero** del deadlock (*deadlock detection & recovery*), che non applica nessuna tecnica per impedire l'occorrere di una situazione di stallo, ma fornisce degli algoritmi per la sua rilevazione ed eliminazione

E se nessuna di queste strategie ha successo? Il sistema si arenerebbe nello stato di deadlock, con un conseguente progressivo deterioramento delle prestazioni dovuto al blocco indefinito delle risorse che impediscono l'evoluzione dei processi. Spesso in questi casi l'unica soluzione è il riavvio manuale del sistema operativo.
Vediamo ora più in dettaglio le varie strategie di gestione.

Prevenzione del deadlock

Prevenire il deadlock significa fare in modo che almeno una delle quattro condizioni per cui si verifica sia soddisfatta. Studiamo i casi separatamente.
Bisogna anzitutto verificare che il principio di *mutua esclusione* sia applicato solo su quelle risorse intrinsecamente non condivisibili, come ad esempio una stampante; estenderlo anche a risorse condivisibili aumenta solo il rischio di


Questo/a opera è pubblicato sotto una Licenza Creative Commons.

insorgere in situazioni di stallo. Un esempio di risorse intrinsecamente condivisibili sono i file in sola lettura, cui l'accesso simultaneo può tranquillamente essere garantito a tutti i processi che ne avessero bisogno.
Per evitare invece la condizione di *possesso e attesa* si deve fare in modo che ogni volta che un processo chiede risorse non ne possegga già qualcun'altra. A tale scopo si possono usare due tecniche:

- un processo chiede e ottiene tutte le risorse prima di iniziare l'esecuzione. Se ad esempio un processo deve modificare mille file e infine stamparli, accederà a tutte le risorse all'avvio (stampante compresa) e ne manterrà il possesso per tutta la durata della computazione e solo quando termina le rilascerà. Se il sistema operativo non riesce a dargli tutti gli accessi, il processo non viene nemmeno attivato. Lo svantaggio di questa soluzione non è da poco. Può infatti passare molto (troppo) tempo prima che il processo usi tutte le risorse da lui bloccate all'apertura, mantenendo inutilmente altri processi in attesa rallentando di conseguenza il parallelismo del sistema
- un processo che possiede alcune risorse e vuole chiederne altre deve prima rilasciare tutte quelle che già detiene, quindi richiedere l'accesso a quelle che gli servono più quelle che aveva già. Una volta ottenute, il processo esegue le computazioni opportune e quindi le rilascia a disposizione di altri. Pur guadagnando in parallelismo rispetto alla soluzione precedente, il limite di tale tecnica è che bisogna strutturare opportunamente il programma tenendo conto che una risorsa (e il suo contenuto) terminato l'uso possa essere modificata da altri processi, e questo per un programmatore non è un compito banale

In entrambi i casi ho un basso utilizzo delle risorse, che possono essere assegnate ma rimanere inutilizzate per un periodo lungo; talmente lungo che potrebbe avvenire una starvation.
Se *non applicare la pre-emption* soddisfa una delle quattro condizioni di deadlock, imporla ove possibile scongiura gli stalli. Quando è possibile? Quando il rilascio anticipato non crea problemi di consistenza delle informazioni.
Come nel caso precedente, possiamo utilizzare due tecniche:

- se un processo detiene delle risorse e ne chiede altre per le quali deve attendere, allora deve rilasciare anticipatamente tutte le risorse possedute, che vengono aggiunte in una lista di tutte quelle per cui il processo sta aspettando. Il processo sarà fatto ripartire solo quando potrà ottenere tutte le risorse, vecchie e nuove. Siamo sostanzialmente nello stesso approccio dell' *hold & wait*, con la differenza che qui non devo attendere il termine d'uso della risorsa da parte del processo dato che gli viene imposto
- se un processo chiede risorse si verifica prima di tutto che siano disponibili, nel qual caso gli viene dato l'accesso. Altrimenti se sono già associate a un altro processo che però non le usa perché sta attendendo altre risorse, viene imposto a quest'ultimo di rilasciarle anticipatamente e vengono aggiunte alla lista delle risorse per cui il processo richiedente è in attesa. Se invece le risorse sono già associate a un altro processo che le sta usando, deve attendere che questi le rilasci. Riassumendo, si applica la pre-emption solo su processi in attesa di altre risorse, altrimenti no.

Va considerato che questi protocolli non possono essere applicati indistintamente su tutti i tipi di risorse, ad esempio sarebbero impensabili su stampanti o unità nastro.
Quarta e ultima condizione da scongiurare per prevenire il deadlock è l' *attesa circolare*. La tecnica per prevenirlo è forse la più contorta vista finora, e richiede che sia dato un ordinamento globale univoco su tutti i tipi di risorsa R_i, in modo che sia sempre possibile sapere se una è maggiore o minore di un'altra rispetto all'indice *i*'. Tale indice è un numero naturale che dovrebbe essere assegnato con un certo criterio, rispettando i comuni ordini d'uso delle risorse; ad esempio è plausibile che un file sia utilizzato prima di una stampante, dunque ha senso che il suo indice sia inferiore a quello della periferica.
L'ordinamento di per sé non previene il deadlock, bisogna applicare la seguente politica, divisa in due punti:

- se un processo chiede un certo numero di istanze della risorsa R_j e detiene solo risorse R_i con $i<j$, se le istanze sono disponibili gli vengono assegnate, altrimenti dovrà attendere;
- se invece richiede istanze della risorsa R_j e detiene risorse R_i con $i>j$, allora il processo dovrà prima rilasciare tutte le sue risorse R_i, quindi richiederle tutte, vecchie e nuove comprese.

Ad esempio se il processo P detiene la risorsa R_{25}, potrà chiedere solo risorse indicizzate da R_{26} in su, e non da R_{25} in giù, a meno che non rilasci tutte quelle che ha già. Questa tecnica impedisce che più processi possano attendersi l'un l'altro: gli indici lo impediscono.
Normalmente applicare questi protocolli di controllo è responsabilità degli sviluppatori di applicazioni, ma esistono alcuni software che svolgono il lavoro per loro. Questi programmi sono noti come *Witness*.

Evitare il deadlock

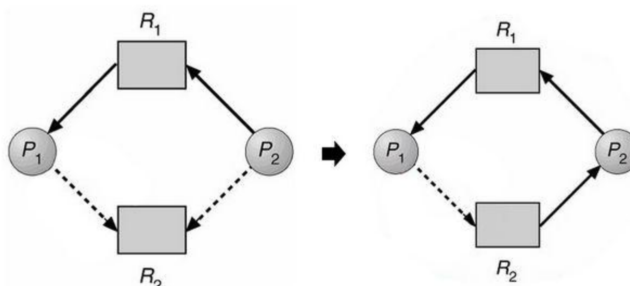
http://www.swappa.it

Questo/a opera è pubblicato sotto una Licenza Creative Commons.

La *deadlock prevention* applica alcune regole su come devono essere fatte le richieste di accesso alle risorse, garantendo che almeno una delle quattro condizioni perché si abbiano stalli non sia verificata. Il problema è che sono regole molto restrittive, che determinano un basso rendimento del sistema e di utilizzo delle periferiche. Una possibile alternativa è verificare a priori se la sequenza di richieste e rilasci di risorsi effettuate da un processo porta a stalli, tenendo conto delle sequenze dei processi già accettati nel sistema. Ciò significa **evitare il deadlock**, e richiede che i processi forniscano al sistema operativo alcune informazioni sul loro comportamento futuro, così che questo possa sapere in qualsiasi momento:

- il numero massimo di risorse per ogni processo
- quante risorse sono assegnate e quante sono disponibili
- la sequenza delle richieste e dei rilasci da parte di un processo

Una volta ottenute queste informazioni, il che non è sempre semplice, bisognerà implementare un algoritmo che ne tenga conto e metta in moto una serie di procedure perché il sistema non entri mai in uno stato di deadlock.
Introduciamo a tal proposito il concetto di **stato sicuro**.
Uno stato si dice *sicuro* se il sistema operativo può allocare le risorse richieste da ogni processo in un certo ordine,*garantendo* che non si verifichi deadlock. Attenzione però, essere in uno stato *non sicuro* non significa essere certi di arrivare in una situazione di stallo in futuro, ma non poter assicurare il contrario; tanto basta perché sia considerato non sicuro. Più formalmente, uno stato è sicuro soltanto se esiste una **sequenza sicura**, ovvero una sequenza di processi P_1, P_2, ... , P_n tale che le richieste che ogni processo P_i può fare possono essere soddisfatte dalle risorse attualmente disponibili più tutte le risorse detenute dai processi P_j, con j<i. In altre parole, è una sequenza di processi tale che le richieste di ognuno possono essere soddisfatte con le risorse già disponibili o in via di rilascio dai processi precedenti ad ogni singolo processo i-esimo. Si tratta quindi di un ordine con cui andare a considerare le richieste dei processi, garantendo così di non avere attese circolari.
Facciamo un esempio. Se ho un processo P_i che ha bisogno di alcune risorse che non sono immediatamente disponibili, dovrà attendere che tutti i P_j che vengono prima di lui abbiano rilasciato le proprie. Quando ciò accade, P_i può ottenere finalmente tutte le risorse necessarie, completare la sua esecuzione e infine rilasciarle. Una volta terminato P_i, il processo P_{i+1} potrà ottenere le sue risorse e così via. Se si soddisfa tale sequenza, lo stato del sistema è sicuro.
Compito degli algoritmi che si occupano di evitare situazioni di stallo è continuare ad accertarsi che il sistema, all'avvio nativamente in stato sicuro, permanga in tale stato ad ogni richiesta di allocazione delle risorse. Dato che ciò può comportare una maggiore tempo di attesa per i processi, lo sfruttamento delle risorse potrebbe essere più basso del suo potenziale.
Studiamo ora due possibile tecniche per implementarlo.
Variante del grafo di allocazione delle risorse
Se abbiamo un sistema di allocazione delle risorse con un'unica istanza per risorsa, per individuare i deadlock si può utilizzare una versione modificata del *grafo di allocazione delle risorse*, in cui oltre agli archi di richiesta e assegnazione c'è anche l' **arco di prenotazione**. Rappresentato come una freccia tratteggiata, indica quali risorse saranno prima o poi necessarie ad un processo perché possa portare avanti la propria computazione; quest'informazione deve essere data prima che inizi l'esecuzione. Rilevare un ciclo sugli archi di prenotazione significa che un processo in futuro potrebbe fare una richiesta che porta il sistema in una situazione di stallo, il che va evitato.
Esempio:



Nel grafo di allocazione delle di risorse di sinistra notiamo come sia P1 che P2 prenotino entrambi la risorsa R2, che mette un'unica istanza a disposizione. Se, come vediamo nel grafo di destra, R2 concede l'accesso a P2 cadiamo in uno

Questo/a opera è pubblicato sotto una Licenza Creative Commons.

stato non sicuro, dal momento che basta che P1 faccia anche lui richiesta ad R2 perché si chiuda il ciclo e si abbia una situazione di stallo. La richiesta di P2 deve dunque essere rifiutata dal sistema.

Algoritmo del banchiere

Il limite del grafo di allocazione delle risorse è che non può essere usato se ho risorse con istanze multiple, perché l'esistenza di un ciclo non implica necessariamente un deadlock. L' **algoritmo del banchiere** permette invece di gestire tali situazioni, pur essendo in media meno efficiente data la sua maggiore complessità computazionale. L'idea che sta dietro l'algoritmo è che il sistema operativo prima di concedere delle risorse a un processo, gli chiede il numero massimo di istanze di cui avrà bisogno; se verifica che l'allocazione lo lascia in uno stato sicuro, allora la concede, altrimenti il processo dovrà attendere finché il sistema non ritenga di avere abbastanza risorse per farlo progredire. Bisogna dunque che siano soddisfatte due ipotesi iniziali:

- ogni processo deve dichiarare a priori il numero massimo di istanze di cui avrà bisogno, o il sistema gli rifiuta la richiesta. Proprio come un banchiere, che deve sapere quanti soldi dare a un cliente
- ogni processo dovrà restituire in un tempo finito le risorse utilizzate. Poi della durata del processo in sé all'algoritmo del banchiere non importa nulla, l'importante è che prima o poi rilasci le risorse. Che ci metta molto o ci metta poco, nemmeno questo importa: l'unica cosa che si vuole è evitare il deadlock, a qualunque prezzo.

Una delle maggiori complessità dell'algoritmo del banchiere è che deve gestire parecchie strutture dati.

Dati **m** numero delle risorse ed **n** numero dei processi, avrò:

- **available[1..m]**, le risorse disponibili. Ogni i-esima posizione indica il numero di istanze di tipo i disponibili
- **max[1..n, 1..m]**, una matrice che definisce il numero massimo di risorse che un processo può chiedere. Se max[i][j] è uguale a *k*, allora il processo P_i può chiedere al più *k* istanze di risorse del tipo R_j
- **allocation[1..n, 1..m]**, una matrice che dice il numero di risorse di ogni tipo attualmente assegnate a ogni processo. Se allocation[i][j] è uguale a *k*, allora il processo P_i detiene attualmente *k* istanze di risorse del tipo R_j
- **need[1..n, 1..m]**', una matrice che specifica il numero di risorse che il processo dovrà ancora richiedere nella sua vita. In particolare, need[i][j] = max[i][j] - allocation[i][j]

Queste strutture possono variare nel tempo sia in dimensione che in valore.

L'algoritmo di distingue in due parti, una che verifica la sicurezza ed uno per la richiesta delle risorse.

L'algoritmo **di sicurezza** consiste nella verifica dello stato sicuro, deve cioè trovare - se esiste - una sequenza sicura. Di seguito lo pseudocodice, coi commenti *sotto* ogni passo:

Work[1..m]

\\ *un vettore che rappresenta quali sono le richieste di risorse*

Finish[1..n]

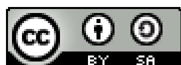
\\ *un vettore che rappresenta quali processi hanno completato la computazione, e quindi hanno rilasciato a un certo punto le risorse*

1. **Work = Available; Finish[i] = false per i = 0, 1, ... , n-1**
   \\ *fase di inizializzazione: segno come disponibili gli elementi del vettore delle risorse, e assegno valore negativo per ogni processo del vettore Finish (dato che non hanno ancora terminato la computazione)*
2. **Si cerca i tale che**
   - **Finish[i]==false**
   - **$Need_i$ <= Work**

Se non esiste tale i, vai al passo 4

scandisco Finish finché trovo un processo non ancora terminato e che richiede un numero di risorse inferiore a quelle indicate nel vettore Work

3. **Work = Work + Allocation[i]; Finish[i] = true**
   Vai al passo 2
   \\ *rappresenta il caso in cui posso soddisfare la richiesta. Una volta che il processo ha terminato la computazione e rilasciato le risorse, andrò ad aggiornare il vettore delle allocazioni disponibili, sommandogli le risorse che erano state assegnate al processo i-simo (ormai rilasciate)*
4. **Se, per ogni i, Finish[i]==true, allora lo stato è sicuro**
   \\ *notare che l'algoritmo può richiedere m x n^2 operazioni prima di arrivare a questo passo*


Questo/a opera è pubblicato sotto una Licenza Creative Commons.

Fatta questa verifica, l'algoritmo prosegue con la gestione delle richieste. Come prima, lo pseudocodice è commentato *sotto* i vari passi.

Request[i]

\\ *rappresenta la richiesta del processo Pi in un certo momento*

1. **Se Request[i]<=Need[i], vai al passo 2**
   Altrimenti solleva errore: il processo ha ecceduto il numero massimo di richieste
   \\ *se il numero di richieste è minore (o al più uguale) al numero massimo di richieste che saranno effettuate da i, allora vai al passo 2. Se lo supera, allora il processo aveva cannato a fornire al sistema il numero massimo di richieste che avrebbe chiesto*
2. **Se Request[i]<=Available, vai al passo 3**
   Altrimenti Pi deve attendere: risorse non disponibili
   \\ *se il numero di richieste è minore o uguale a quelle disponibili in quel momento, allora posso passare al passo 3, in cui tali richieste saranno soddisfatte.*
3. **Si ipotizzi di stanziare le risorse richieste**
   - **3.1 Available = Available - Request[i]**
   - **3.2 Allocation[i] = Allocation[i] + Request[i]**
   - **3.3 Need[i] = Need[i] - Request[i]**

Se lo stato risultante è sicuro, al processo Pi vengono confermate le risorse assegnate. Altrimenti Pi deve aspettare per le richieste Request[i] e viene ristabilito il vecchio stato di allocazione delle risorse.

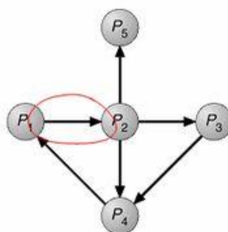
la sicurezza dell'assegnamento viene valutata in tre punti: in 3.1 viene diminuito il numero di risorse disponibili in base alle richieste del processo; in 3.2 viene incrementato il numero di risorse assegnate al processo; in 3.3 viene decrementato il numero di risorse del processo che ancora devono essere soddisfatte. Se queste allocazioni mi lasciano in uno stato sicuro (e lo verifico con l'algoritmo di verifica visto prima) confermo tutto, altrimenti non posso dare le risorse al processo perché entrerei in uno stato non sicuro. In questo secondo caso riporto tutto alla situazione precedente all'allocazione e lascio che il processo attenda.

Rilevazione e ripristino del deadlock

Senza algoritmi per prevenzione o evitare il deadlock, tale situazione può verificarsi. Sistemi di questo tipo devono mettere a disposizione algoritmi per *rilevarne la presenza dopo che sono avvenuti*, e per *ripristinare una situazione di corretto funzionamento* eliminando lo stallo. Vedremo come questi requisiti possano adattarsi a sistemi con istanze singole o multiple delle risorse.

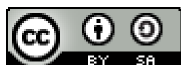
Rilevazione

Abbiamo già imparato che nel caso in cui tutte le risorse abbiano un'unica istanza, il grafo di allocazione delle risorse ci torna sempre molto utile. Utilizzeremo in questo caso una variante ridotta, il **grafo di attesa** (o *wait for*), dal quale sono stati rimossi tutti i nodi che rappresentavano le risorse e gli archi si trovano ora a collegare due processi. La loro nuova interpretazione è che un certo processo P_i richiederà una risorsa posseduta da un altro processo P_j, quale risorsa non ci interessa. Ad esempio nel *grafo di attesa* seguente il processo P1 vuole accedere a qualche risorsa detenuta da P2.



Il sistema dovrà dunque mantenere aggiornato il grafo di attesa dei suoi processi ed eseguire periodicamente l'algoritmo di rilevazione di eventuali cicli, che segnalano l'occorrenza di un deadlock. Cosa si intende per periodicamente? Si può decidere di farlo partire ogni volta che un processo chiede una risorsa e scopre che è occupata, ma data la frequenza dell'evento avrei un overhead troppo elevato. Un'altra soluzione potrebbe essere far partire l'algoritmo ad intervalli di tempo prefissati; è vero che tra un intervallo e l'altro potrebbero essersi verificati degli stalli, ma è sicuramente meno costoso che eseguirlo ad ogni richiesta fallita.

Se invece nel sistema sono presenti risorse con istanze multiple, bisogna cambiare il tipo di algoritmo di rilevamento, molto simile a quello del banchiere.



Questo/a opera è pubblicato sotto una Licenza Creative Commons.

Work[1..m]
Finish[1..n]

1. **Work = Available; per i = 0, 1, ... , n-1**

se Allocation[i] != 0, allora Finish[i] = false;
altrimenti Finish[i] = true;

1. **Si cerca i tale che**
   - **Finish[i] == false**
   - **$Request_i$ <= Work**

Se non esiste tale i, vai al passo 4

2. **Work = Work + Allocation[i]; Finish[i] = true**
   Vai al passo 2
3. **Se Finish[i] == false per qualche i, con 0 = i < n, allora si ha deadlock**
   Inoltre, se Finish[i] == false, allora il processo P_i è in deadlock

Vanno fatte anche in questo caso le considerazioni sulla frequenza di esecuzione dell'algoritmo di rilevamento viste prima per i grafi d'attesa.

Ripristino

Quando un algoritmo di rilevazione riscontra effettivamente una situazione di stallo, può comportarsi in due modi: informa l'utente della situazione e lascia a lui il compito di gestirla manualmente, oppure fa in modo che sia il sistema operativo ad occuparsi del ripristino automatico. Quest'ultimo può adottare come soluzione la *terminazione forzata di un processo* o il *rilascio anticipato delle risorse*, che andremo a vedere maggiormente in dettaglio.

Esistono due strategie per ripristinare il sistema dal deadlock **terminando** i processi coinvolti, entrambe che comportano il rilascio delle risorse assegnate ai processi abortiti. Esse sono

- *abortire tutti i processi in deadlock*, che interromperà sì il deadlock, ma a caro prezzo dato che vengono persi tutti i risultati parziali che i processi terminati avevano calcolato fino a quel punto. R' funzionale, ma comporta uno spreco consistente di risorse computazionali
- *abortire un processo alla volta fino a eliminare il ciclo di deadlock*, che causa per contro un overhead considerevole, dal momento che ad ogni processo abortito sarà da eseguire l'algoritmo di rilevazione del deadlock (per capire quando fermarsi). A ogni modo, esistono diversi criteri su cui basarsi per scegliere l'ordine di eliminazione dei processi. Ad esempio potrei considerare le loro priorità, da quanto tempo sono in esecuzione o quanto tempo gli manca perché terminino, quante e quali tipi di risorse stanno usando o devono ancora richiedere, quanti processi devono essere terminati o ancora se il processo è interattivo o no.

Se invece si volesse eliminare il deadlock usando il **rilascio anticipato delle risorse**, bisognerebbe deallocare forzatamente certe risorse a determinati processi, passandone poi l'accesso ad altri, e continuare a farlo fino a quando viene superata la situazione di stallo. Perché ciò avvenga senza problemi, si devono tener conto di tre aspetti fondamentali:

- **scelta della vittima**, ovvero quali risorse e quali processi devono essere interrotti. Anche in questo caso vengono considerati diversi criteri per la selezione, molto simili a quelli visti per la terminazione
- **rollback**. Se si forza un processo a rilasciare le risorse che sta usando, ovviamente non lo si mette più nelle condizioni di continuare la sua normale esecuzione. Bisognerebbe quindi riportarlo (*rollback*) ad un certo stato sicuro e farlo ricominciare da lì. Ciò implica che il sistema operativo debba tener traccia di più informazioni sui processi, e questo non accade molto spesso. Per questo motivo molto spesso il rollback è *totale*, ovvero abortisce il processo e lo fa ripartire dall'inizio
- **starvation**. Dato che la selezione della vittima è basata soprattutto su fattori costo, può succedere che in caso di deadlock sia sempre lo stesso processo ad essere abortito. Non potendo progredire si arena in una situazione di *starvation*, risolvibile considerando nel fattore costo anche il numero di rollback imposti.

http://www.swappa.it

Questo/a opera è pubblicato sotto una Licenza Creative Commons.