SISTEMI

I processi

Prof. Viglietti Francesco

Classe 4 B info

A.S. 2010-11

Processi

Un OS consiste in un gran numero di attività che vengono eseguite più o meno contemporaneamente dalla CPU e dai dispositivi presenti in un computer. Senza un modello adeguato, la coesistenza delle diverse attività sarebbe difficile da descrivere e realizzare. Il modello che è stato realizzato a questo scopo prende il nome di modello concorrente ed è basato sul concetto astratto di processo.

Processo è un'attività controllata da un programma che si svolge su un processore. Ad ogni istante un processo può essere descritto da:

1. immagine in memoria centrale
2. immagine nel processore
3. stato di avanzamento

Lo stato di avanzamento di un processo descrive la situazione in cui il processo si trova. Infatti i processi hanno in genere un avanzamento discontinuo e durante la loro evoluzione subiscono varie transizioni di stato in base agli eventi che si verificano.

Stati di un processo - 1

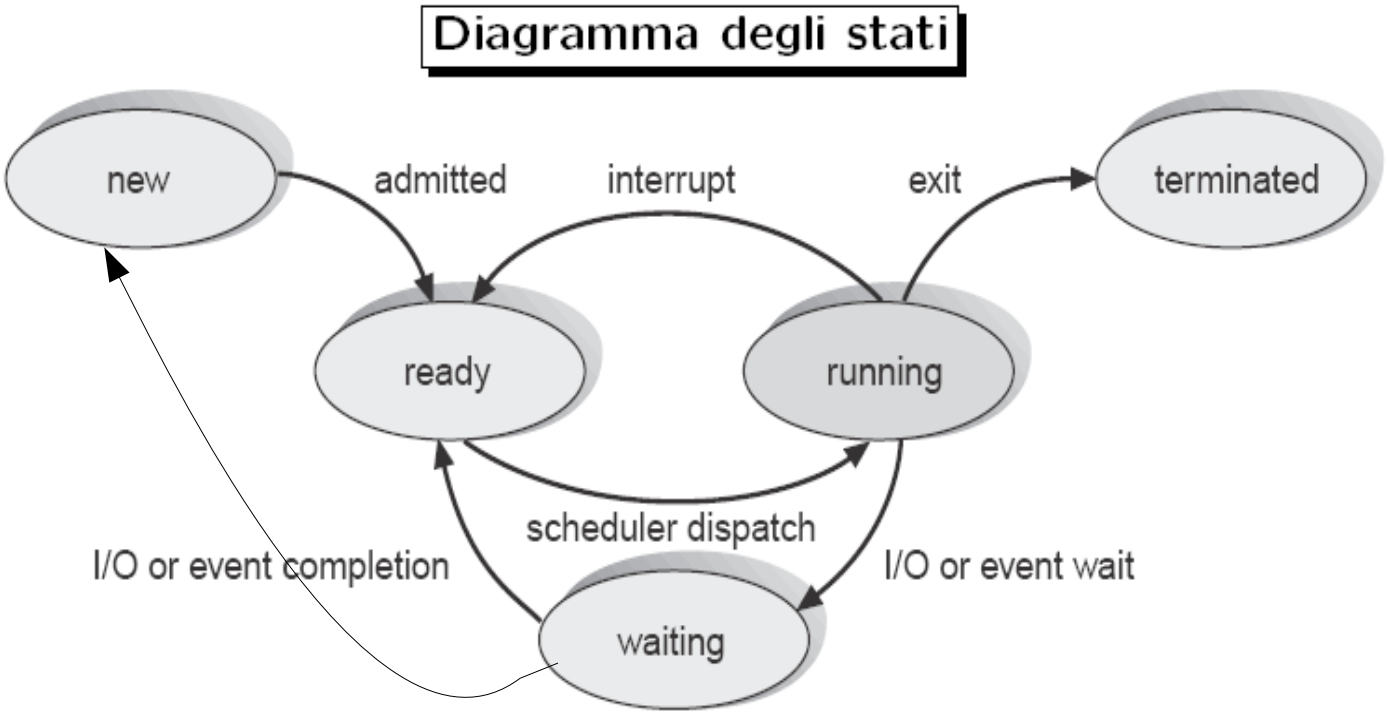
New (disponibile**)**: quando è stata richiesta l'esecuzione del programma che si trova su disco

Ready (pronto): quando il processo è stato caricato in memoria centrale e aspetta gli venga assegnato una CPU

Running (esecuzione): quando al processo viene assegnata la CPU. È il solo stato d'avanzamento

Waiting (attesa): quando il processo ha richiesto un sevizio al OS e ne attende il cc

Terminated (termina zione): quando il processo termina l'es cuzione e può rilasci re le risorse usate

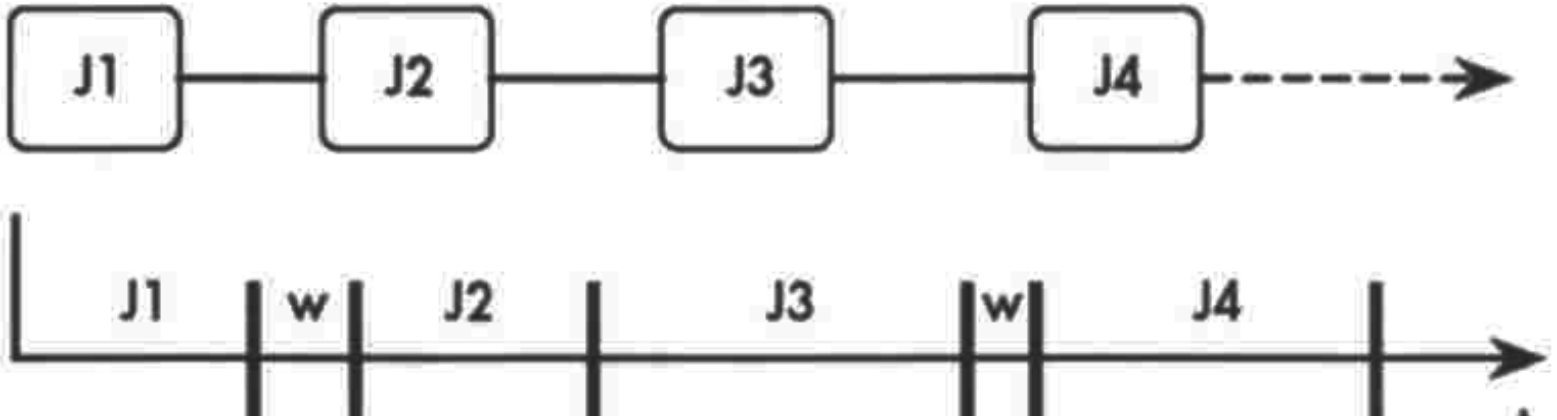


Stati di un processo - 2

I processi passano da uno stato all'altro per vari motivi:
New → Ready: quando e caricato in memoria centrale mediante una politica di scheduling.
Ready → Running: assegnazione della CPU
Running → Terminated: quando il processo termina
Running → Waiting: quando il processo richiede un operazione del OS o un'operazione di I/O
Waiting → Ready: quando finisce l'esecuzione della richiesta esterna del processo
Running → Ready: nella modalità time-sharing, quando termina lo slice time assegnato al processo

Processi sequenziali

Il modo più semplice che il sistema può realizzare per assicurare una corretta esecuzione è prevedere processi sequenziali nei quali tutti i job vengono eseguiti rigorosamente nello stesso ordine in cui sì trovano nel programma originale e le risorse vengono acquisite secondo l'ordine di richiesta.
In questo modo il tempo totale di esecuzione del procedimento completo è uguale alla somma dei tempi di esecuzione dei singoli job: vantaggiose sono la semplicità e la linearità dell'esecuzione, mentre gli svantaggi riguardano i lunghi tempi di inattività a cui possono essere costrette le risorse, oppure le attese dei singoli job prima di acquisire la loro risorsa e diventare sottoprocessi.

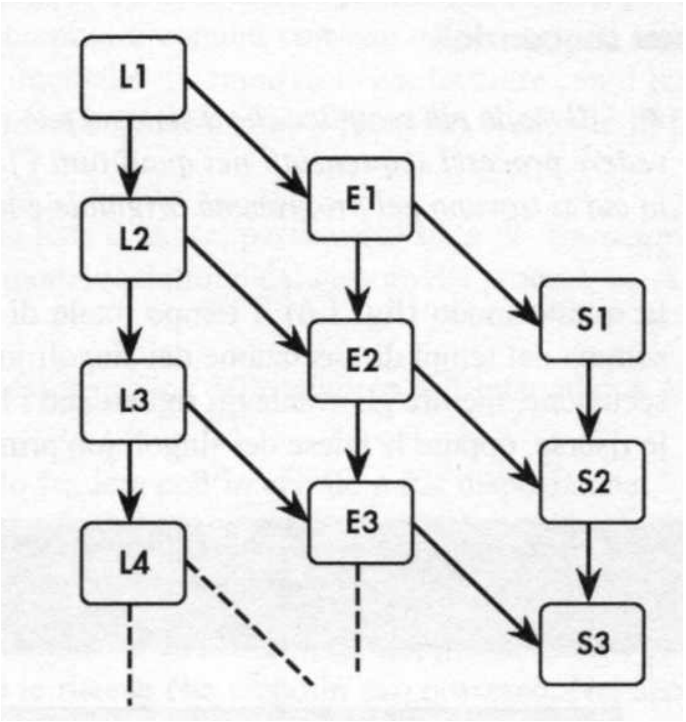


Processi non sequenziali

Se prevediamo un processo che richiede 3 cicli di operazioni, possiamo dire che il dispositivo di lettura verrà richiamato dai job L1, L2, L3, mentre l'unità di calcolo servirà solo ai job E1, E2, E3 e infine i job S1, S2, S3 richiederanno l'unità disco. Mentre una classe di risorsa è in uso, le altre sono ferme; è quindi possibile ottimizzare l'uso di queste risorse.

In questo caso si prevede un'organizzazione del sistema in processi non sequenziali. Nei processi di questo tipo i job non vengono eseguiti nell'ordine in cui sono indicati, tuttavia vengono salvaguardate alcune condizioni che garantiscono la correttezza dei risultati. Potremo eseguire E2 prima di S1 ma non prima di L2. Il diagramma delle precedenze, indica la precedenza che si deve garantire tra i job del processo. I processi non sequenziali ordinati secondo diagrammi di precedenza non consentono un'esecuzione casuale dei singoli job, ma allargano il concetto di esecuzione sequenziale garantendo che certi gruppi di operazioni siano comunque eseguite in ordine sequenziale, ma che le operazioni legate a risorse distinte possano anche essere eseguite contemporaneamente: in questo modo il tempo totale di esecuzione dell'intero processo diventa anche inferiore alla somma dei tempi d'esecuzione dei singoli job.

Diagramma delle precedenze



Parallelismo

Una tecnica che prevede la realizzazione di processi non sequenziali come quella appena illustrata nei diagrammi di precedenza si chiama programmazione parallela e può essere definita utilizzando i tre seguenti costrutti linguistici:

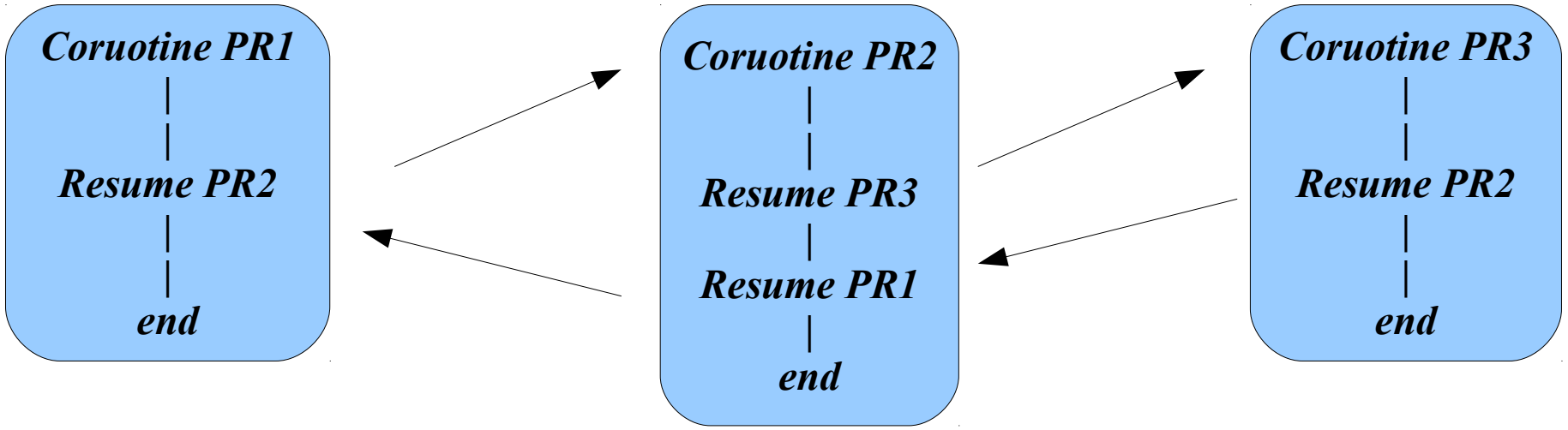
coroutine;

fork/join;

cobegin/coend.

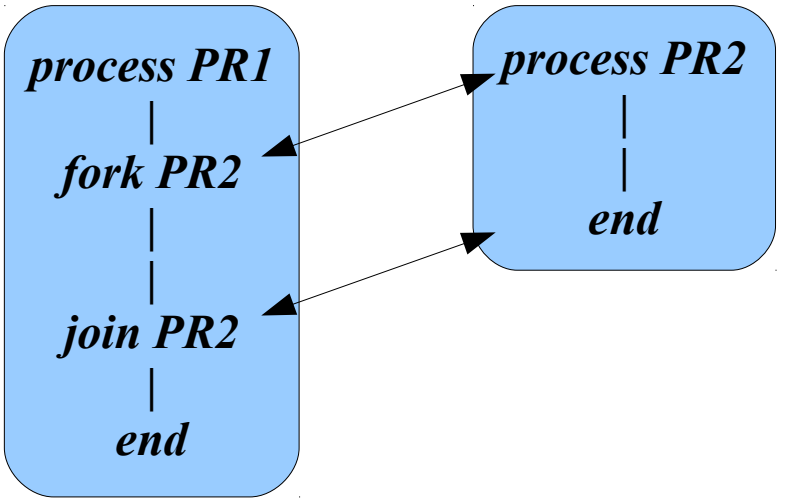
Coroutine

La Coroutine è una specifica che va indicata nei processi che vanno eseguiti in parallelo. L'istruzione **resume** consente il passaggio del controllo da una coroutine a un'altra, ma la differenza rispetto ai normali salti a procedura è che non c'è l'obbligo di un ritorno del controllo dalla procedura chiamata alla chiamante. Quando durante l'esecuzione s'incontra un'istruzione di **resume**, il controllo passa alla coroutine indicata e la precedente si blocca: In figura 3 coroutine interagenti. La **resume** nella PR1 comporta un salto alla PR2 e uno stop all'esecuzione della PR1; la coroutine PR2 viene eseguita finché non s'incontra una **resume** verso la PR3: la seguente **resume** causa un ritorno a PR2 che continua da dove si era bloccata e così via. La coroutine non realizza una vera programmazione parallela, ma semplicemente un interleaving, cioè una esecuzione alternata tra più task, una specie di illusione di esecuzione parallela.



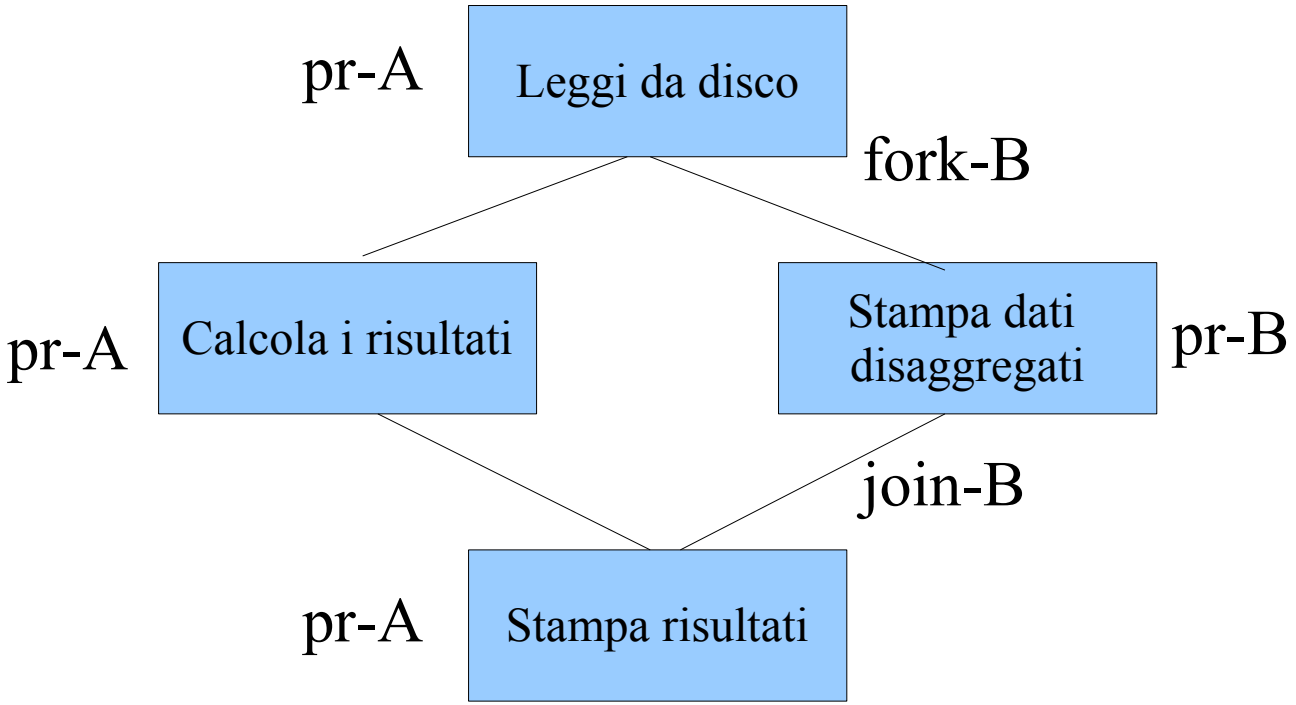
Fork / Join 1

Il costrutto fork/join, a differenza del precedente, realizza una esecuzione parallela dei processi ed è presente in diversi OS (Linux). L'istruzione fork comporta l'attivazione del processo indicato in modo parallelo rispetto al processo chiamante; l'esecuzione torna a essere sequenziale quando il processo chiamante presenta l'istruzione join: a questo punto se il processo chiamato è terminato, il chiamante continua il suo normale lavoro, in caso contrario il chiamante dovrà attendere il termine del processo chiamato.
Il processo PR1 viene eseguito sino all'istruzione fork PR2; adesso i due processi PR1 e PR2 vengono eseguiti in parallelo. Il processo PR1 si ferma quando incontra l'istruzione join PR2 ed eventualmente attende il termine del processo PR2 se questo non ha già concluso il suo lavoro. A questo punto il processo PR1 prosegue indisturbato sino alla fine.
Un 'applicazione tipica del costrutto fork/join riguarda le operazioni che coinvolgono periferiche distinte.



Fork / Join 2

Supponendo di dover leggere dati da un file su disco e di dover effettuare complessi calcoli riepilogativi e valutazioni statistiche su di essi e di dover riportare il tutto sulla stampante, possiamo pensare di definire una struttura che preveda la lettura dei dati e l'esecuzione in parallelo delle operazioni di calcolo e della stampa dei dati disaggregati. Terminate le operazioni di stampa dei dati originali (pr-B), probabilmente i calcoli da svolgere (pr-A prima del join B) saranno finiti; allora la stessa stampante potrà essere utilizzata per accodare agli altri dati appena emessi i risultati delle valutazioni effettuate. In questo modo abbiamo evitato alla stampante un lungo periodo di attesa e garantito all'utente un tempo di esecuzione complessiva più ridotto.



Cobegin/Coend

L'ultimo costrutto è il cobegin/coend, che permette l'esecuzione parallela di singole istruzioni. La sintassi generale è la seguente:

```
Program PR;
Istruzione I0;
cobegin
I1; 12; ...;In;
coend
end.
```

In essa è possibile indicare le singole istruzioni che devono essere eseguite in modo parallelo. Nello schema, tutte le istruzioni I0, In contenute tra le parole chiave cobegin e coend saranno eseguite parallelamente. La differenza rispetto al costrutto precedente è rilevante poiché si parla di parallelizzazione di singole istruzioni all'interno di un unico processo e non più di esecuzione contemporanea di parti di processi distinti.
Quali che siano il modo e il costrutto con cui i processi possono interagire, resta da affrontare il problema più grande: la sincronizzazione. Non interessa solo il legame che sussiste tra i processi, cioè quali regole il sistema operativo deve seguire nell'assegnazione delle risorse richieste, ma è fondamentale la gestione della programmazione concorrente.

Esecuzione concorrente

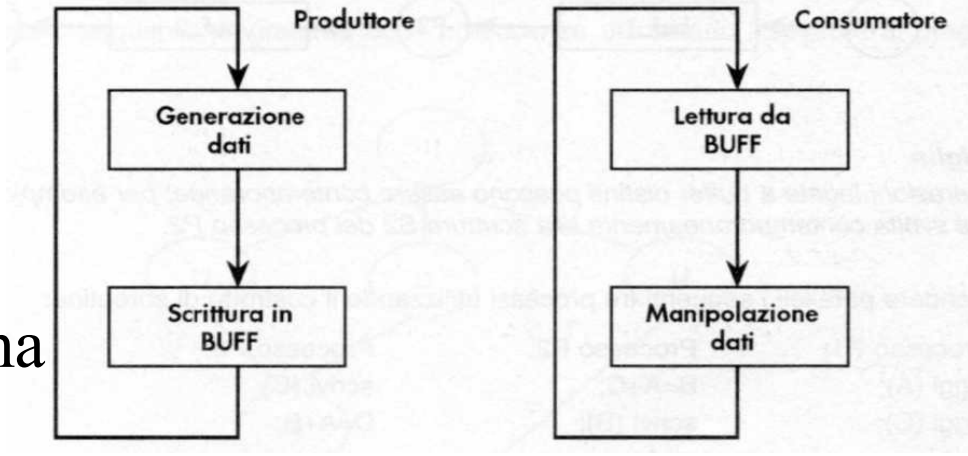
La differenza tra programmazione parallela e programmazione concorrente; spesso risulta superflua, tuttavia, più propriamente, il problema della programmazione concorrente riguarda i processi che si trovano a competere (**concorrere**) per la proprietà di una stessa risorsa: ecco quindi che per programmi concorrenti si intendono quei processi che richiedono la stessa risorsa nello stesso istante, perciò il sistema dovrà utilizzare tecniche e strumenti per regolare le assegnazioni. Vediamo i 3 tipi di relazione in cui si possono trovare dei processi concorrenti:

cooperazione;
competizione;
interferenza.

Due task sono in **cooperazione,** se sono legati logicamente; per esempio, uno dei due ha bisogno, per evolvere, di dati prodotti dall'altro. Il caso classico è quello della relazione Produttore-Consumatore: un task P genera dati che vengono inseriti in un'area buffer (BUFF), e un secondo task C legge i dati dall'area BUFF e li manipola per conseguire risultati finali. In breve, si deve fare in modo che C legga da BUFF solo dopo che P vi abbia inserito alcuni dati. Questa operazione può essere ripetuta ciclicamente, ma deve essere eseguita rigorosamente nell'ordine detto.

Esempio cooperazione

Processo Produttore Consumatore... sequenza corretta P-C-P-C. Se viene eseguita inavvertitamente la sequenza P-P-C-C, il risultato sarà che avremo manipolato 2 volte il secondo blocco di dati perdendo il primo.



Competizione:
Due processi in competizione, sono indipendenti dal punto di vista logico, tuttavia la scarsità delle risorse del sistema genera una dipendenza reciproca durante il lavoro. Siamo nel caso in cui i due processi hanno in comune solo la necessità di lavorare con una determinata risorsa; in questo modo essi si trovano a competere per la sua acquisizione e i tempi di completamento del lavoro possono dipendere sensibilmente dalla velocità nel loro confronto diretto. Un esempio di questo tipo di problema è il caso della mutua esclusione nella quale l'acquisizione di una risorsa è esclusiva di un processo, con una conseguente attesa di tutti gli altri processi richiedenti. Questo fenomeno può essere fonte di grossi problemi per il sistema e deve essere strettamente controllato.

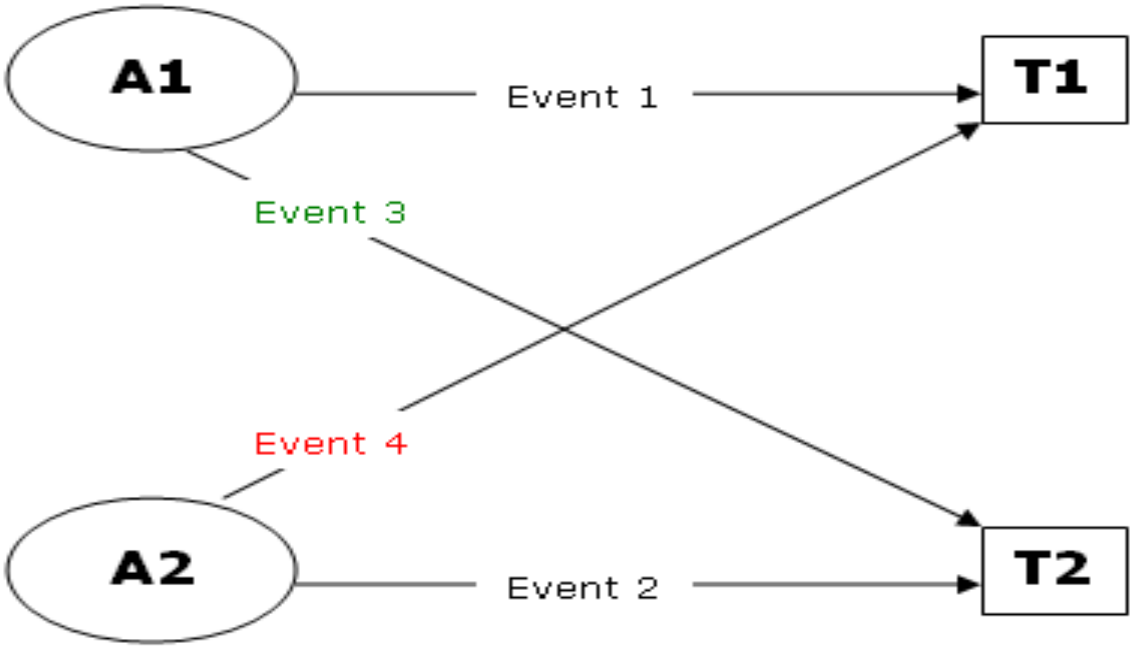
Interferenza

L'interferenza è una degenerazione della competizione. Il sintomo si riscontra quando il risultato dei nostri processi dipende dal momento in cui essi vengono eseguiti: cioè se il risultato dipende dalla sequenza temporale di utilizzo di certe risorse del sistema da parte di tutti i processi che le richiedono. Si deve fare in modo che i processi prendano la risorsa di cui hanno bisogno solo in istanti precisi, secondo modi definiti e non la rilascino in momenti critici del loro lavoro; nessun processo deve strappare in modo inconsulto una risorsa ad altri processi. L'unica garanzia è di sincronizzare ogni operazione sulle risorse a rischio senza basarsi sulla "velocità" di acquisizione. Se i processi acquisiscono le loro risorse in modo disordinato, è possibile che essi lavorino su dati manipolati da altri e che quindi, facendo eseguire il programma più volte, si ottengano sempre risultati diversi: gli errori di questo tipo sono difficilissimi da identificare e da correggere poiché sintatticamente il lavoro è corretto.

Concorrenza esempi.... deadlock

Siano T1 e T2 due risorse, A1 e A2 due processi che devono accedere a T1 e T2 contemporaneamente, prima di poter terminare il programma. Supponiamo che OS assegni T1 ad A1, e T2 ad A2. I due processi sono bloccati in attesa circolare.(un processo aspetta che termini l'altro per poter prendere la risorsa, e viceversa) Si dice che A1 e A2 sono in **deadlock.** Il deadlock è una condizione da evitare, è definitiva e nei sistemi reali, se ne può uscire solo con metodi "distruttivi", ovvero uccidendo i processi, riavviando la macchina, etc. Il deadlock è un problema che coinvolge tutti i processi che utilizzano un certo insieme di risorse.

Esiste anche la possibilità che un processo non possa accedere ad una risorsa perché "sempre occupata".



Concorrenza esempi.... starvation

Sia R una risorsa, siano P1, P2, P3 tre processi che accedono periodicamente a R. Supponiamo che P1 e P2 si alternino nell'uso della risorsa. P3 non può accedere alla risorsa, perché utilizzata in modo esclusivo da P1 e P2. Si dice che P3 è in **starvation**.

A differenza del deadlock, non è una condizione definitiva; è possibile uscirne adottando opportune politiche di assegnamento, ma è comunque una situazione da evitare.

Sezioni critiche

Le azioni atomiche vengono compiute in modo indivisibile. Soddisfano la condizione: o tutto o niente.
Nel caso di parallelismo apparente: l'avvicendamento (context switch) fra processi avviene prima o dopo l'azione (no interferenza). Il meccanismo degli interrupt (su cui è basato il context switch) garantisce che un interrupt venga eseguito prima o dopo un'istruzione, mai "durante".
Nel caso di parallelismo reale: si garantisce che l'azione non interferisca con altri processi durante la sua esecuzione. Anche se più istruzioni cercano di accedere alla stessa cella di memoria, la politica di arbitraggio del bus garantisce che una delle due venga servita per prima.
Se le sequenze di istruzioni non vengono eseguite in modo atomico, come possiamo garantire la non interferenza?
L'idea generale è che dobbiamo trovare il modo di specificare che certe parti dei programmi devono essere eseguite in modo atomico.
La parte di un programma che utilizza una o più risorse condivise viene detta sezione critica (CS). Dobbiamo garantire che le CS siano eseguite in modo mutualmente esclusivo (atomico) ed evitare situazioni di blocco, sia dovute a deadlock sia dovute a starvation.

I semafori introduzione

Un'interruzione comporta il salvataggio dello stato della macchina e quindi del task in corso; in ogni caso l'interruzione di un lavoro non può avvenire in un momento arbitrario, ma si dovrà assicurare al sistema il completamento del singolo passo del processo in corso in modo che, se ciò sarà possibile, il sistema sia in grado di consentire un successivo proseguimento del lavoro.

Quando tra processi c'è il pericolo che si generi un rapporto di competizione, è necessario provvedere alla costruzione di meccanismi di sincronizzazione.

Il modo più semplice e sicuro sarebbe quello di consentire a un processo di prendere una risorsa e stringerla in pugno per tutta la durata del lavoro. Questo metodo fuga la maggior parte dei pericoli di accesso inconsulto alle risorse, tuttavia, non è conveniente per un sistema di elaborazione. La competizione, in realtà, può essere legata a passi specifici del programma che vanno preservati da interruzioni e interferenze indesiderate: queste "parti" di programma che possono generare conflitti si chiamano zone critiche. Alcuni esempi di zona critica possono essere le operazioni di stampa (non è ragionevole interrompere una stampa con un'altra stampa), oppure generiche operazioni di scrittura di dati su di un supporto di memorizzazione (non possiamo consentire la lettura di un file, mentre è in atto una modifica del suo contenuto: ne va dell'integrità dei suoi dati), o ancora le operazioni di scrittura in un'area buffer quando non è ancora terminata la lettura del blocco di dati scritto nel buffer precedentemente.

primitive

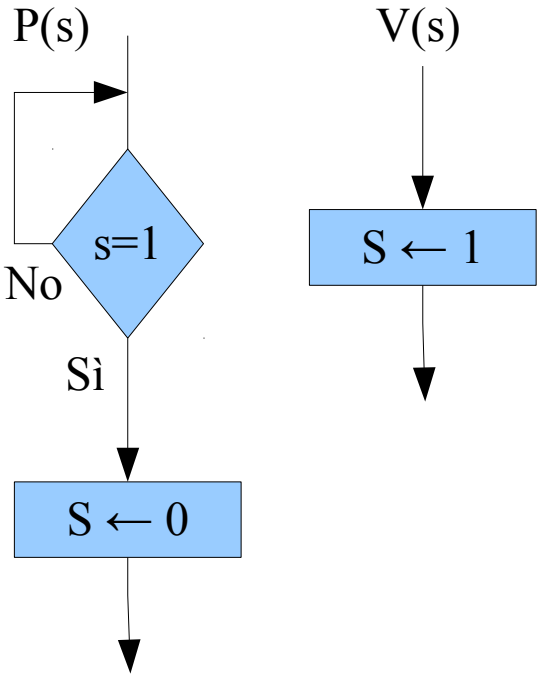
In breve, si deve fare in modo che queste operazioni siano rese indivisibili, cioè non interrompibili, così che quando la risorsa viene acquisita non possa essere rilasciata prima della fine di queste operazioni a rischio.

La tecnica base di gestione di una zona critica si basa sul principio del semaforo e utilizza due procedure elementari (o primitive di sincronizzazione) a basso livello. Queste due procedure utilizzano una variabile il cui valore viene sempre controllato prima di accedere alla risorsa critica: se la variabile dà il via, la risorsa può essere allocata e il task può proseguire; in caso contrario, il task si porrà in attesa. Il "semaforo" è rappresentato dalla variabile binaria s che funge da parametro per le due procedure P(s) e V(s): se "s=0" indica che la risorsa critica a cui si vuole disciplinare l'accesso è occupata e quindi non si può far altro che attendere che questa venga rilasciata; se la variabile semaforo "s=1", allora la risorsa relativa è libera e disponibile per l'assegnazione.

Esempi primitive

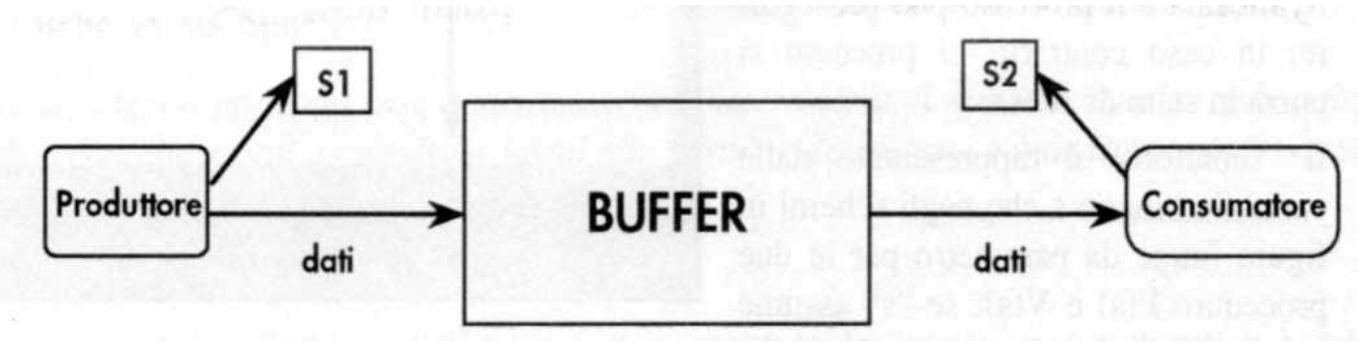
La procedura P(s) viene richiamata dal task che vuole impegnare una risorsa; se, "s=1", allora la risorsa è libera, quindi viene allocata al task richiedente e il semaforo viene impostato al valore "0". Se dal test risulta, invece, che il semaforo ha valore "0", allora la procedura ripete il controllo finché non riscontra il valore "1". Questa tecnica può essere migliorata in modo da non costringere il task a ripetere indefinitamente l'operazione di test sulla variabile s.

La procedura V(s) viene richiamata da un processo che vuole rilasciare una risorsa in suo possessi La procedura provvede semplicemente a impostare la variabile al valore "1" (che indica la avvenuta disponibilità della risorsa) e a incaricare il sistema operativo del rilascio della risorsa collegata.



Produttore/consumatore

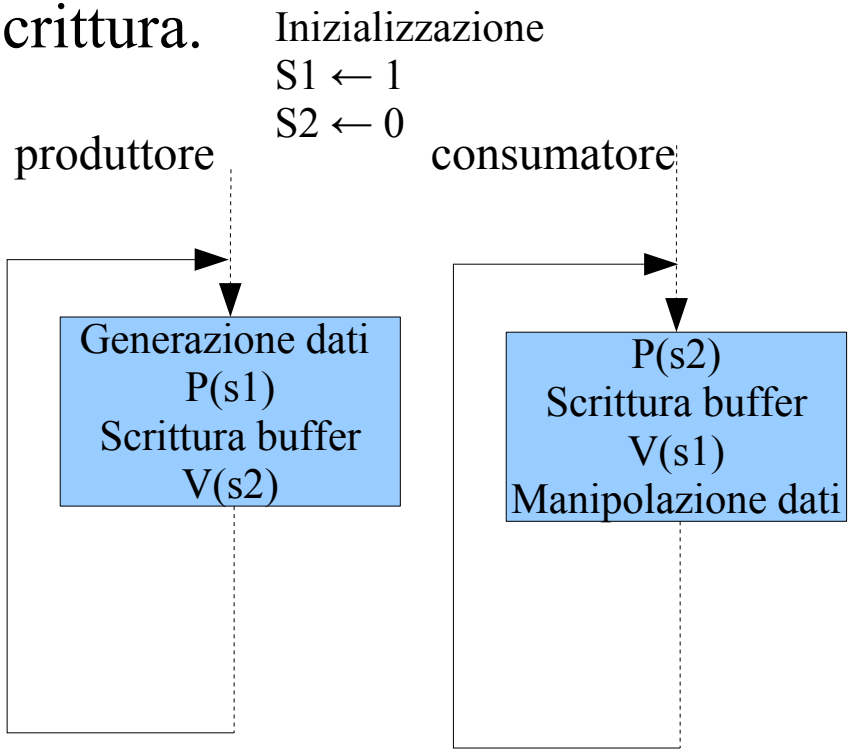
Supponiamo di avere, due processi che intendono lavorare in modo esclusivo su un'unica area buffer: il primo (Produttore) produce dati e li scrive in un buffer; il secondo (Consumatore) legge i dati che il primo processo ha inserito nel buffer e vi esegue operazioni. Una soluzione classica prevede l'uso di due semafori s1 e s2, che regolano le due diverse operazioni sullo stesso buffer comune.



Dallo schema notiamo che non risulta sufficiente un solo semaforo visto che, oltre a garantire l'accesso esclusivo alla risorsa, si deve anche far sì che non si tenti di leggere dal buffer prima che ne sia terminata la scrittura al suo interno. Per questo motivo si è pensato di realizzare il semaforo s1, che controlla l'accesso in scrittura al buffer, e il semaforo s2, che controlla l'accesso in lettura. La procedura di inizializzazione garantisce che il primo processo ad accedere al buffer sia il produttore, poiché indica la disponibilità in scrittura (s1=1) e la chiusura in lettura (s2=0).

Produttore/consumatore

Il Produttore genera dati e impegna la risorsa (s1 passa a 0 con l'esecuzione di P(sl)); alla fine del lavoro provvede ad aprire la possibilità di lettura mantenendo bloccata la scrittura. In questo modo si consente a un eventuale Consumatore di attingere al contenuto del buffer, ma non è possibile per un secondo Produttore scrivere nuovi dati sopra quelli attuali. Il Consumatore, una volta che è riuscito a ottenerne il permesso di accesso, legge dal buffer e poi, prima di passare alle operazioni di manipolazione dei dati appena acquisiti, libera l'accesso in scrittura.

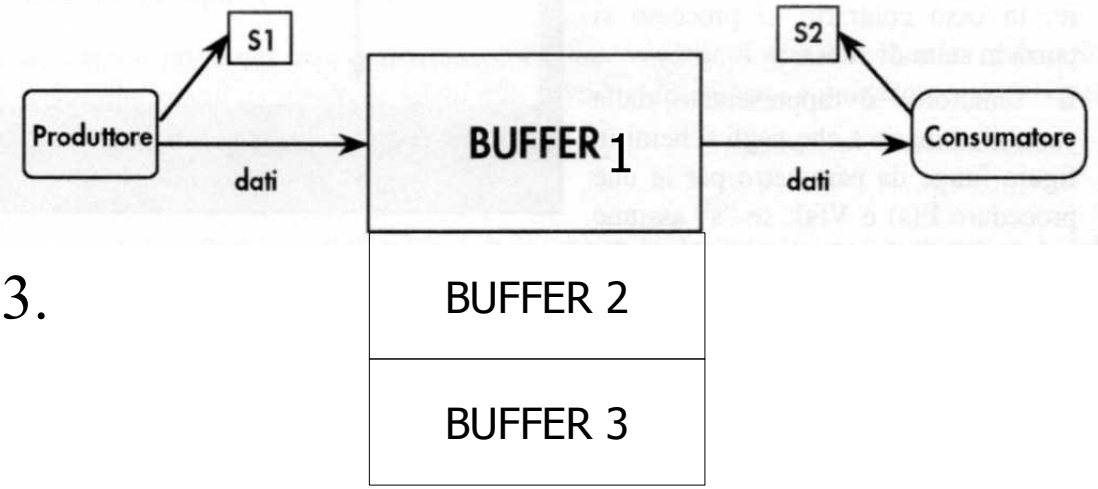


La scelta di liberare l'operazione di scrittura (con la V(sl)) prima di eseguire la manipolazione dei dati consente al sistema nel Suo complesso di guadagnare tempo poiché, mentre sono in corso operazioni sui dati appena letti, ne possono essere inseriti nuovi nello stesso buffer, senza attendere il termine delle operazioni del consumatore.

Semaforo generalizzato

Se il sistema deve gestire una risorsa che non è presente in un unico esemplare, ma che, tuttavia, è oggetto di contesa, allora si può ricorrere a una tecnica basata sul semaforo generalizzato. La variabile semaforo non è più binaria, ma è un semplice contatore delle unità disponibili della risorsa; se il suo valore è 0 allora nessuna unità della risorsa sarà disponibile; in caso contrario è possibile effettuare un'allocazione. Il numero massimo che può assumere la variabile semaforo deve coincidere con il numero totale di unità presenti nel sistema. A questo valore dovrà essere inizializzato il semaforo che sarà sottoposto a decremento.

Abbiamo una struttura molto simile alla precedente, la sola differenza è nel numero dei buffer: in questo caso se il produttore è molto veloce, allora può scrivere in uno dei buffer liberi mentre il consumatore sta leggendo da un altro buffer. Poiché abbiamo 3 buffer a disposizione, allora i 2 semafori avranno il valore massimo 3.



Esempio semaforo generalizzato

Supponiamo che il nostro sistema sia dotato di 4 stampanti simili e che consenta il lavoro di 8 persone in multiutenza: il sistema non è in grado di soddisfare più di 3 richieste di stampe contemporaneamente. Un semaforo generalizzato impostato al valore "4" consente la regolarizzazione degli accessi ai 3 dispositivi.

Anche in questo caso si utilizzano le procedure P(s) e V(s) con le varianti seguenti. La procedura P(s) contiene un test che controlla il valore del "contatore": se s è positivo allora ci sono ancora delle unità libere; l'allocazione comporta un decremento del numero di unità ancora disponibili.

La procedura V(s) che riguarda il rilascio della risorsa contiene semplicemente l'operazione di incremento del numero di unità disponibili.

Precisiamo che le procedure P(s) e V(s) sono semplicemente le operazioni di base per la gestione dei meccanismi di sincronizzazione degli accessi.