Il processo di I/O generato seguirà le stesse transizioni fin qui descritte, con l'unica differenza che il processore interessato sarà il processore incaricato delle operazioni di I/O (un canale di I/O oppure lo stesso processore centrale).

Una volta giunto a terminazione, il processo di I/O richiama il controllore del traffico che si preoccuperà di eliminare il descrittore del processo di I/O e di riportare in stato di pronto il processo che l'aveva generato.

I SISTEMI A MULTIPROCESSORE

Come abbiamo visto, molto spesso per migliorare il rendimento di un calcolatore si può ricorrere a sistemi che utilizzano al loro interno più processori, in grado di effettuare più elaborazioni in parallelo.

Fino a qualche tempo fa nei sistemi di elaborazione erano presenti un solo processore centrale (*master*) e diversi processori (*coprocessori o processori slave*) con funzioni specifiche.
Oggi invece si costruiscono calcolatori in cui tutti i processori presenti all'interno sono in grado di esplicare le stesse funzioni e, quindi, ciascuno di essi può svolgere sia le funzioni di master sia di slave.
Ciò consente di ridurre il sovraccarico di un processore, poiché qualunque altro può eseguire le routine di sistema operativo. Permane, però, il collo di bottiglia relativo all'accesso alla memoria ancora condivisa tra i vari processori, tanto che deve essere previsto un apposito dispositivo (*arbitro*) per gestire la competizione per l'accesso alla risorsa.

Questo problema risulta ridimensionato con l'uso di reti locali di computer, in cui ad ogni processore sono associate una propria memoria centrale ed un certo numero di periferiche, e l'accesso alla memoria comune avviene solo quando occorre gestire l'interazione tra processi residenti su processori diversi.

LA PROGRAMMAZIONE CONCORRENTE

Quando si osserva un sistema multiprogrammato in time sharing che ogni pochi millisecondi assegna il processore a processi diversi, si può certamente pensare che i processi coinvolti avanzino *contemporaneamente*, anche se sappiamo che in realtà le strategie attuate dal gestore dei processori stanno procedendo ad una *sequenzializzazione* delle esecuzioni.

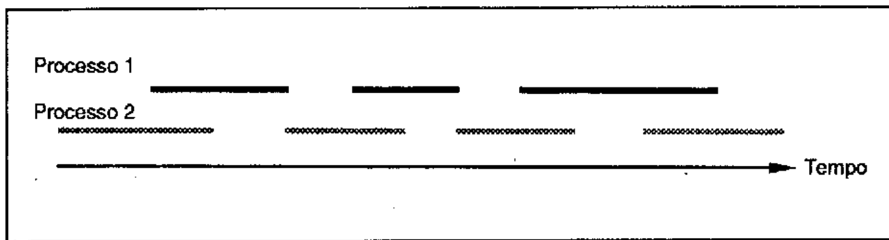
Esistono però casi in cui, grazie a dispositivi intelligenti, è possibile attuare un vero parallelismo poiché alcune funzioni possono essere svolte indipendentemente dal processore centrale: è il caso dei processori dedicati di DMA che, ricevute alcune indicazioni iniziali, attivano la trasmissione dati da periferica a memoria o viceversa lasciando il processore master libero di dedicarsi ad altri processi.
Questa linea di tendenza ha portato, grazie all'abbassamento dei costi dell'hardware, all'affermazione di calcolatori multiprocessor che mettono a disposizione del si-

stema operativo più processori di pari flessibilità e potenza, o addirittura di calcolatori con *architettura parallela* in cui più processori *concorrono* all'esecuzione di uno stesso programma.

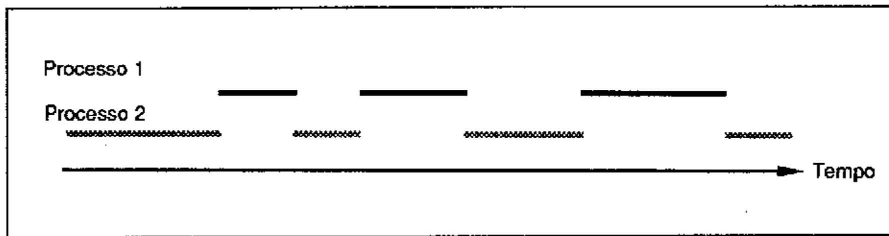
In questo contesto, *due processi si dicono* ***concorrenti*** *se esistono contemporaneamente* o, più formalmente *se uno viene creato prima che l'altro termini*

Più precisamente, la concorrenza tra due azioni può esprimersi:

- come un *avanzamento alternato* nel tempo (*interleaving*), quando il numero delle operazioni da compiere è superiore al numero degli esecutori, come nel caso di una singola CPU che deve rispondere a richieste di più processi.
  In questo caso siamo di fronte sì a un "finto" parallelismo, ma è sempre possibile immaginare di avere a disposizione più esecutori distinti (*esecutori virtuali*) e ipotizzare che le azioni vengano svolte in *modo asincrono* l'una rispetto all'altra;



- come una reale *sovrapposizione* delle operazioni nel tempo (*overlapping*), e ciò non può che richiedere due o più distinti esecutori.



In quest'ultimo caso ovviamente si dovranno utilizzare dei *linguaggi di programmazione concorrenti* (Modula-2, ADA, Concurrent Pascal, ...) che, oltre a fornire precise tecniche e notazioni specifiche per esprimere e gestire questo potenziale parallelismo, devono permettere di risolvere gli eventuali problemi di sincronizzazione e comunicazione tra processi.

In particolare, sarà indispensabile riuscire ad individuare, nella descrizione di un programma, le attività che possono essere eseguite concorrentemente e possedere una notazione per contraddistinguerle in un programma concorrente.

In quest'ottica, il costrutto linguistico che utilizzeremo per descrivere esecuzioni concorrenti verrà costituito dalla notazione *cobegin coend* che ha il compito di racchiudere una o più istruzioni o procedure da eseguire in modalità concorrente:

```
cobegin
  P1(...);
  P2(...);
  .....
  PN(...)
coend;
```

Naturalmente, l'ordine di esecuzione di questi processi e l'alternanza di assegnazione della CPU da un processo all'altro per eseguire le singole istruzioni *non è prevedibile* e potendo cambiare esecuzione per esecuzione rendono il controllo dell'esecuzione stessa estremamente problematica.
Quando tutte le procedure sono terminate, il programma principale riprende dall'istruzione posta dopo il **coend** e i singoli statement vengono valutati sequenzialmente da un singolo processore.

✍ Per esempio, consideriamo, passo per passo, il calcolo di una delle due radici di un'equazione di 2° grado: $ax^2+bx+c=0$

$$x_1 = \frac{-b + \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

Si avrebbe:

sequenzialmente

```
begin
  T1:=-b;
  T2:=b*b;
  T3:=4*a;
  T4:=T3*c
  T5:=T2-T4;
  T6:=sqrt(T5);
  T7:=2*a
  T8:=T1+T6;
  X:=T8/T7
end;
```

concorrentemente

```
cobegin
  T1:=-b; T2:=b*b; T3:=4*a; T7:=2*a
coend
T4:=T3*c
T5:=T2-T4;
T6:=sqrt(T5);
T8:=T1+T6;
X:=T8/T7
```

Anche questo semplice esempio può evidenziare il vantaggio in termini di tempo di esecuzione, poiché le 4 istruzioni comprese tra il **cobegin** e il **coend** potrebbero essere eseguite contemporaneamente anche da 4 processori diversi.

COMPETIZIONE E COLLABORAZIONE TRA I PROCESSI

Il problema della sincronizzazione tra processi deriva, come abbiamo già accennato, anche dalla necessità di condividere le risorse in generale limitate di un sistema di elaborazione.

La ripartizione di queste richieste, infatti, necessita di un coordinamento che assicuri la sequenza corretta delle operazioni.

All'inizio dell'unità didattica abbiamo visto come lo schedulatore dei lavori, quando genera un processo da inviare in stato di pronto, gli assegni *staticamente* tutte le risorse *non condivisibili*, risorse che coincidono con quelle la cui assegnazione dinamica produrrebbe una sovrapposizione di effetti tale da rendere errata l'interpretazione delle informazioni.

> ✍ Immaginiamo, per esempio, di assegnare a turno dinamicamente la stessa stampante a due processi in esecuzione. Se i processi richiedono solo saltuariamente l'uso di tale risorsa per stampare una riga, potrebbe capitare che le righe di stampa appartenenti ai due processi vengano mescolate tra loro sullo stesso foglio di carta.

Con *l'assegnazione statica* questo problema viene risolto, ma nasce un nuovo inconveniente: quando un processo molto lungo utilizza la risorsa solo in poche occasioni, questa risulta non disponibile per molto tempo agli altri processi, anche se in realtà rimane quasi sempre inattiva.

Questo problema si può risolvere facendo assegnare staticamente dallo schedulatore dei lavori *solo le risorse* richieste esplicitamente dall'utente, mentre sarà compito del gestore del traffico assegnare tutte le altre risorse, comprese quelle non condivisibili, per le quali l'assegnazione avverrà ancora staticamente, ma solo a partire dal momento in cui il processo ne richiede l'uso per la prima volta.

La risoluzione di questo inconveniente può essere estesa affrontando formalmente tutta la categoria di problemi legata ***all'interferenza*** e alla ***cooperazione*** tra processi.

Interferenza e cooperazione

Spesso i processi non possono evolvere in modo del tutto indipendente, ma devono essere sincronizzati per garantire un perfetto funzionamento del sistema.

In particolare, si dice che più processi ***interferiscono*** *tra loro quando, da un punto di vista logico, potrebbero evolvere in modo indipendente se non dovessero usufruire della* ***stessa risorsa****, generando quella che viene detta* ***condizione di corsa*** (*race condition*).

In questo caso, sarà compito del controllore del traffico garantire che, in ogni istante, la risorsa sia assegnata ad uno solo dei processi richiedenti e che ognuno di questi, dopo un tempo più o meno lungo, la possa usare.

Più in particolare, si deve garantire che ogniqualvolta una *risorsa R viene assegnata in modo esclusivo ad un solo processo*, siano soddisfatti i seguenti requisiti:

- ***mutua esclusione***: se un processo P è in esecuzione su una risorsa R nessun altro processo P_i può essere in esecuzione su R;

- ***attesa limitata*** terlo fare in un
- ***avanzamento***: pedire l'access

Si dice invece che du *quando, per evolvere, u*

In questo caso, ovvia tamente dallo stato c quindi, problemi di dai processi per mez

Lo stallo

Come abbiamo visto l'avanzamento dell'

In una situazione d una certa successio scuno in attesa di u caso di ***stallo*** (*dead*

> ✍ Per me Suppon possa r Contem e la pu
>
> **Pro**
> ...
> **rip**
>
> **fin**
>
> Se i te calcol
>
> ①
> ②

- *attesa limitata*: ogni processo P_i che cerca di accedere alla risorsa R deve poterlo fare in un tempo finito (*fairness*);
- *avanzamento*: un processo bloccato in attesa di una risorsa R_1 non deve impedire l'accesso ad una risorsa R da parte degli altri processi P_i (*lockout*).

Si dice invece che due processi *cooperano quando sono logicamente interconnessi, cioè quando, per evolvere, uno dei due deve usare una risorsa che viene* ***prodotta*** *dall'altro*

In questo caso, ovviamente, lo stato di avanzamento di un processo dipende strettamente dallo stato di avanzamento degli altri processi con cui condivide le risorse, quindi, problemi di competizione su tali risorse devono essere risolti *direttamente* dai processi per mezzo di opportune *sincronizzazioni*

Lo stallo

Come abbiamo visto, quando due o più processi competono o collaborano tra loro, l'avanzamento dell'uno dipende da quello degli altri.

In una situazione di questo tipo può allora verificarsi il caso che due processi, per una certa successione di eventi, non possano mai avanzare perché si trovano ciascuno in attesa di un risorsa che deve essere generata dall'altro: si parla in questo caso di *stallo* (*deadlock*).

Per meglio chiarire questa situazione, facciamo riferimento ad un esempio. Supponiamo che un processo A richieda più volte una certa risorsa R e la possa rilasciare solo dopo aver ottenuto una seconda risorsa K.
Contemporaneamente, un secondo processo B richiede più volte la risorsa K e la può rilasciare solo dopo aver ottenuto la risorsa R.

Processo A	**Processo B**
...	...
ripeti	**ripeti**
richiesta di R	richiesta di K
uso di R	uso di K
...	...
richiesta di K	richiesta di R
rilascio di R	rilascio di K
uso di K	uso di R
...	...
rilascio di K	rilascio di R
finché R e K non servono più	**finché** R e K non servono più

Se i tempi di utilizzo delle risorse da parte dei due processi non sono stati ben calcolati, si potrebbe verificare la seguente sequenza di eventi:

1. A richiede ed ottiene l'uso di R;
2. B richiede ed ottiene l'uso di K;

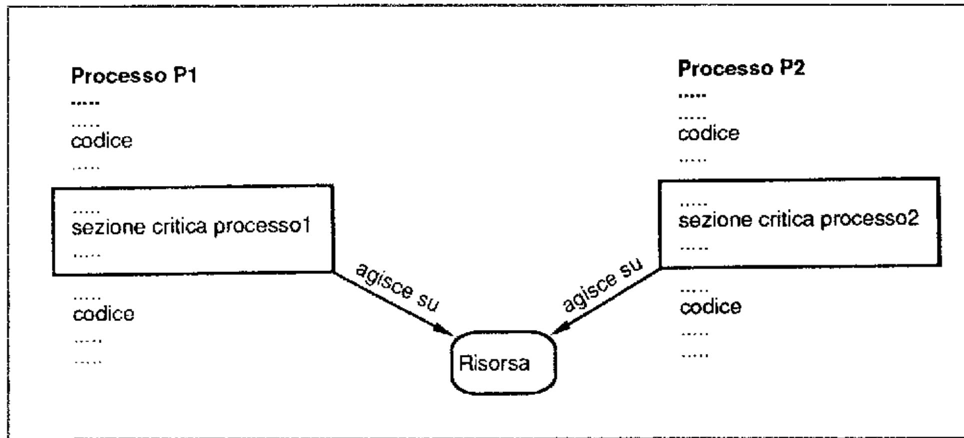
3. A richiede e non ottiene l'uso di K e viene posto in attesa senza che però venga rilasciata R;
4. B richiede e non ottiene l'uso di R (perché ancora assegnata ad A) e viene posto in attesa senza che però venga rilasciata K.

A questo punto A potrebbe tornare in esecuzione, e quindi rilasciare R, solo quando B rilascia K. Ma questo non potrà mai accadere perché B può tornare in esecuzione, e quindi rilasciare K, solo se A gli comunica che ha rilasciato R. I due processi rimangono così bloccati all'infinito in attesa di un evento che non potrà mai accadere.

A questo punto appaiono evidenti le conseguenze che l'accadere di un simile evento può generare in un centro di calcolo, infatti molto spesso questo blocco, inizialmente circoscritto a pochi processi, bloccando le risorse del sistema, crea una serie di stalli sempre più ampi fino al collasso dell'intero centro di calcolo (*deadly embrace - abbraccio mortale*). Esamineremo più avanti questo problema e gli eventuali mezzi per prevenirlo.

LA SINCRONIZZAZIONE TRA PROCESSI

Come abbiamo detto, quando due processi competono per attribuirsi una certa risorsa, sia essa una variabile comune oppure una tabella o un file o ancora un dispositivo fisico, essi *interferiscono*, poiché entrambi *cercano di eseguire un proprio segmento di codice* contenente le istruzioni che permettono di utilizzare detta risorsa.
Si definisce ***sezione critica*** *di un processo quel segmento di codice durante l'esecuzione del quale il processo accede alla risorsa avendo la sicurezza di esserne l'unico utilizzatore*
Appare allora evidente che, in un certo istante, al massimo *un solo* processo può trovarsi in esecuzione della sezione critica relativa ad una certa risorsa.



In questo caso diviene indispensabile disciplinare l'accesso alla sezione critica cercando nel contempo di risolvere i problemi *della mutua esclusione* e di garantire i

vincoli *dell'attesa l*

In questo paragra
riscono, cercherer
efficacemente la s

"Due processi

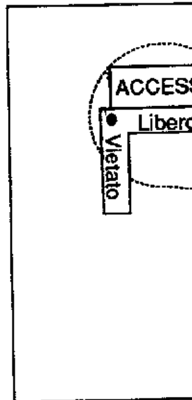
1. una par
2. una pa
sez_criti
cuzione

Occorre garar
esclusione, att

Algoritmo 1

Per cercare di ris
maginando i due
le istruzioni asso
nal computer (ri
lissima anticame
bitro della memo

Una prima soluz
uno schermo su



L'operatore (il p
ca entra e contro

1. se sullo s
cioè l'acc
dolo, fa i