Sistemi Operativi A
Parte II - I sistemi a processi

Augusto Celentano
Università Ca' Foscari Venezia
Corso di Laurea in Informatica

Processi (1)

- Il concetto di *processo* è fondamentale nella teoria dei sistemi operativi
  - il termine è stato coniato negli anni '60 per il sistema operativo MULTICS; è quindi un concetto "antico"
- L'introduzione e il perfezionamento di questo concetto derivano dai problemi osservati in tre modelli di elaborazione
  - batch multiprogrammato
  - time sharing
  - real-time transazionale
- In tutti e tre i casi si osservano due problemi relativi al succedersi di più attività distinte
  - la necessità di preservare lo stato di un'attività prima che abbia terminato il proprio lavoro nel caso di passaggio ad un'altra attività
  - l'ottimizzazione nell'uso delle risorse

Processi (2)

- Il meccanismo fondamentale per gestire l'alternanza di attività è l'interruzione (*interrupt)*, segnalazione di evento che consente all'unità centrale di assumere il controllo in modo definito e privilegiato
- La gestione dell'interrupt deve essere del tutto generale e indipendente dal tipo di attività corrente
- I problemi derivano dalla casualità con cui si presenta l'interrupt rispetto alle attività in corso
  - errori di sincronizzazione
  - errori di interferenza
  - errori di blocco indefinito
  - non determinismo nella successione delle operazioni

Processi (3)

- Un processo è fondamentalmente un programma in esecuzione
  - un'entità che può essere assegnata a un processore ed eseguita su di esso
  - un'unità di attività caratterizzata da un flusso di esecuzione unico e da un insieme di risorse associate (... thread?)
- Formalmente, nella sua formulazione più essenziale, un processo *P* è una coppia di elementi

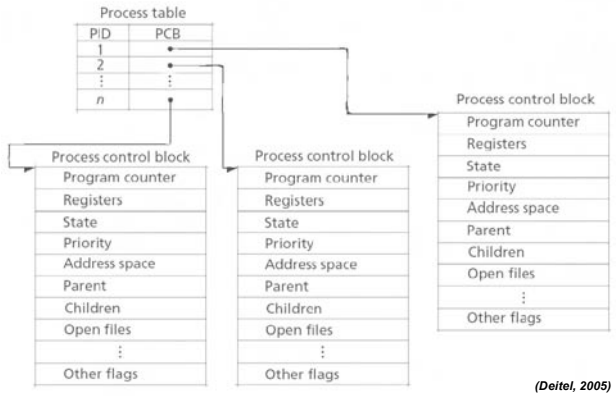
$$P = (C , S)$$

  - C è il *codice* eseguito dal processo (il programma, costante)
  - S è il *vettore di stato* del processo, cioè l'insieme dei dati variabili: valore dei registri, valore delle celle di memoria associate ai dati, stato dei dispositivi di ingresso e uscita, punto in cui si trova l'esecuzione (*program counter*)

Descrittore di processo (1)

- Per gestire più processi, ad ognuno di essi viene associato un *descrittore* (*Process Control Block*)
  - i descrittori contengono le informazioni necessarie per individuare e ripristinare lo stato dei processi
  - ogni descrittore contiene due tipi di informazioni
    - quelle necessarie quando il processo è in esecuzione (ambiente)
    - quelle necessarie quando il processo non è in esecuzione (contesto)
  - i descrittori sono mantenuti dal sistema operativo in aree protette (tabella dei processi)

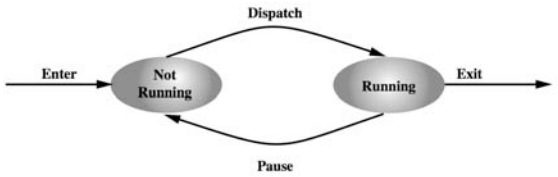
Descrittore di processo (2)



(Deitel, 2005)

Stati di attività un processo (1)

- In una situazione ideale caratterizzata da un processore dedicato, un processo può trovarsi in uno tra due stati: attivo, o in attesa di un evento esterno
  - un processo attivo va in attesa (si sospende) quando chiede un servizio del S.O. (es. una operazione di I/O)
  - un processo in attesa ritorna attivo quando il servizio del sistema operativo è terminato



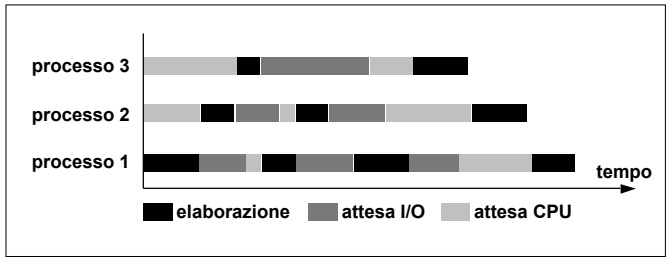
(Stallings, 2005)

Stati di attività di un processo (2)

- In un sistema multiprogrammato vengono eseguiti più processi su un solo processore. La situazione è più complessa
  - un processo può essere logicamente attivo o in attesa di un evento esterno
  - un processo logicamente attivo è in esecuzione quando occupa il processore
  - un processo logicamente attivo può non essere in esecuzione perché il processore non è disponibile (processo inattivo)
- Un solo stato di inattività non è sufficiente
- … oppure: un solo stato di attività non è sufficiente

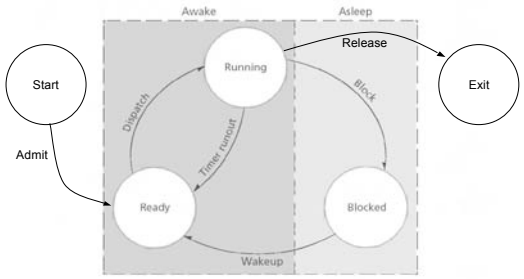
Stati di attività di un processo (3)

- In un sistema multiprogrammato i processi si alternano nell'esecuzione in base a
  - proprio stato di esecuzione (attivo - in attesa)
  - disponibilità di risorse



Un modello a 3+2 stati (1)

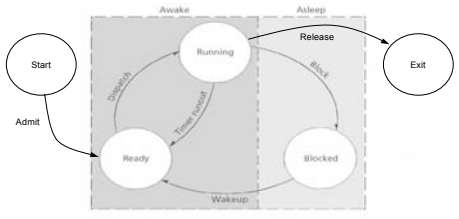
- Per identificare lo stato completo di un processo servono almeno 3 stati
  - attivo, in attesa, pronto
  - + inizio e fine



(Deitel, 2005)

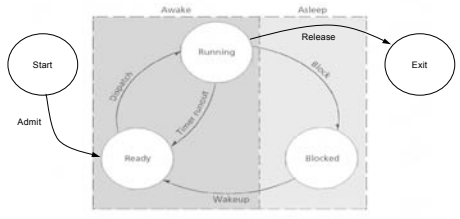
Un modello a 3+2 stati (2)

- *In esecuzione*: un processo che utilizza l'unità centrale
- *Pronto*: un processo che potrebbe essere eseguito se avesse l'uso dell'unità centrale
- *In attesa*: un processo che non può essere eseguito perché richiede che si verifichi un evento esterno



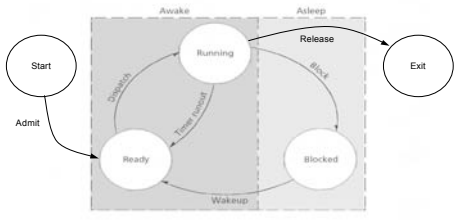
Un modello a 3+2 stati (3)

- *Nuovo*: un processo che inizia l'esecuzione
- *Uscita*: un processo che termina l'esecuzione



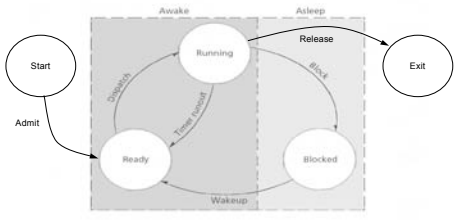
Transizioni di stato (1)

- Un processo in *esecuzione* va in *attesa* (si sospende) quando chiede l'intervento del S.O. (es. per una operazione di I/O)
- Un processo in *attesa* va in stato di *pronto* quando l'evento per cui si era sospeso si verifica



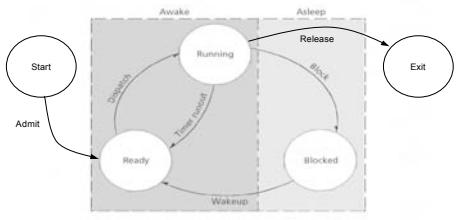
Transizioni di stato (2)

- Un processo *pronto* va in *esecuzione* quando il nucleo gli assegna l'uso dell'unità centrale (*dispatch*)
- Il processo in *esecuzione* va in stato di *pronto* quando il nucleo gli toglie l'uso dell'unità centrale (*timeout*)



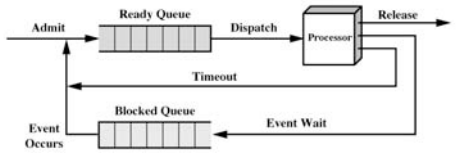
Transizioni di stato (3)

- Un *nuovo* processo viene creato in stato di *pronto*
  - andrà in esecuzione quando gli sarà assegnata l'unità centrale
- Un processo *termina* quando esegue una funzione di terminazione (*exit*) e va nello stato di *uscita*
  - vengono rimosse le risorse occupate



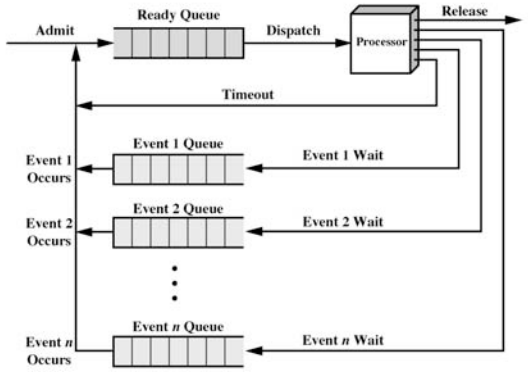
Scheduling dei processi (1)

- I processi pronti sono organizzati in una o più code
  - in base alle politiche di gestione dell'unità centrale
  - la gestione delle code può essere statica o dinamica
- I processi in attesa su dispositivi di I/O normalmente sono organizzati in code, una per ogni dispositivo
  - la gestione delle code è normalmente FIFO



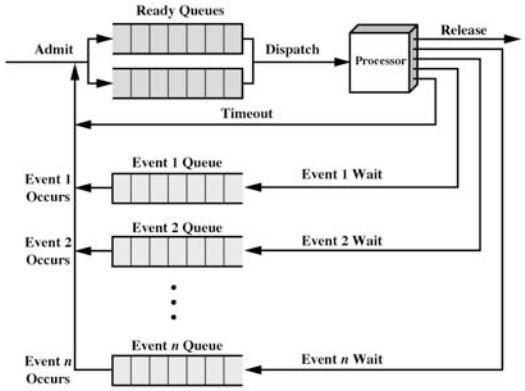
(Stallings, 2005)

Scheduling dei processi (2)



(Stallings, 2005)

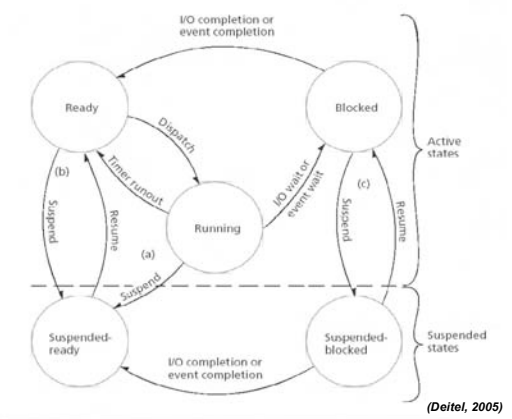
Scheduling dei processi (3)



Gestione degli stati di attesa

- Il processore è più veloce dei dispositivi di I/O
  - tutti i processi in memoria potrebbero essere in attesa di eventi esterni
  - potrebbero esserci altri processi fuori memoria ma in grado di essere eseguiti
- I processi in attesa di I/O lento potrebbero essere portati fuori dalla memoria
  - la memoria si libera per l'esecuzione di altri processi
- Si introduce lo stato di processo sospeso
  - in attesa e sospeso
  - pronto e sospeso

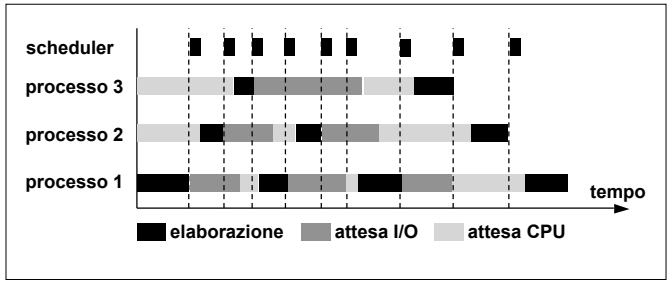
Sospensione dei processi fuori memoria



(Deitel, 2005)

Scheduling dei processi pronti

- La gestione della coda (delle code) dei processi pronti è effettuata da uno *scheduler* a breve termine (*scheduler di CPU*)

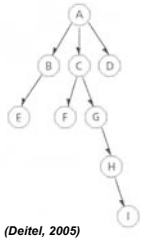


Creazione di un processo (1)

- Assegnazione di un identificatore unico
- Allocazione di memoria per il processo
  - codice
  - dati
- Allocazione di altre risorse nello stato iniziale
  - privilegi, priorità
  - file, dispositivi di I/O
- Inizializzazione del descrittore
- Collegamento con le altre strutture dati del sistema operativo
- Contabilizzazione

Creazione di un processo (2)

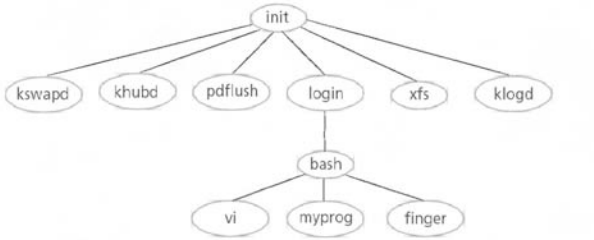
- Un processo può essere creato solo da un altro processo
  - utente
  - di nucleo (del sistema operativo)
- La differenza risiede nella autorizzazioni che il processo creato (*figlio*) eredita dal processo creante (*padre*)
- La creazione di processi può essere iterata a più livelli producendo una struttura ad albero



(Deitel, 2005)

Creazione di un processo (3)

- in Unix tutti i processi nel sistema sono generati a partire da un solo processo iniziale



(Deitel, 2005)

Creazione di un processo (4)

- Relazioni dinamiche con il processo creante
  - il processo padre prosegue
  - il processo padre aspetta
  - il processo figlio non conserva relazioni con il padre (processo *detached*)
- Relazioni di contenuto con il processo creante
  - il processo creato è una copia del processo padre
  - il processo creato esegue un programma diverso

Terminazione di un processo (1)

- Un processo termina con una richiesta al sistema operativo (*exit*) che causa
  - la conclusione delle operazioni di I/O bufferizzate
  - il rilascio delle risorse impegnate (memoria, dispositivi di I/O dedicati)
  - la (eventuale) trasmissione di dati di completamento al processo creante
  - la distruzione del descrittore
- Un processo può terminare per effetto di un altro processo (*kill*), in modo controllato rispetto a privilegi e protezioni
- Un processo può terminare per errore

Terminazione di un processo (2)

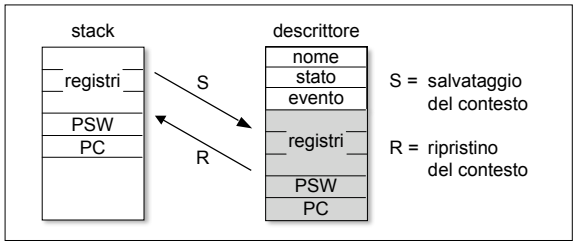
- Le relazioni dinamiche tra processo creante e creato si riflettono sulla terminazione
  - la terminazione di un processo figlio "risveglia" il processo padre in attesa
  - la terminazione di un processo padre causa la terminazione dei processi figli (altrimenti *orfani*)
  - i processi orfani possono essere "adottati" da un altro processo (nonno?)
  - i processi *detached* non sono influenzati dalla terminazione del processo che li ha creati

Commutazione di contesto (1)

- La transizione di stato di un processo è una operazione complessa che, a fronte di una interruzione, modifica il contesto nel quale il processore lavora
- Si assumono le seguenti ipotesi:
  - il verificarsi di una interruzione provoca il salvataggio dei registri del processore (PC, PSW, altri) sullo stack
  - durante il servizio dell'interruzione le interruzioni sono disabilitate
  - il ritorno da una interruzione ripristina i registri del processore dallo stack e riabilita le interruzioni

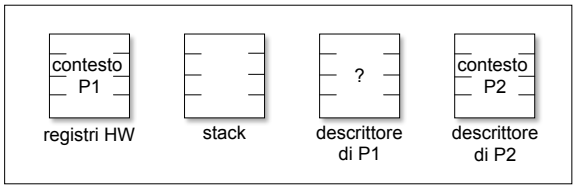
Commutazione di contesto (2)

- La commutazione tra due processi richiede che i loro contesti di esecuzione siano salvati e ripristinati
  - la commutazione avviene solo a seguito di interruzione
  - in cima allo stack c'è il contesto del processo corrente
  - la commutazione può avvenire scambiando informazioni tra lo stack e il descrittore del processo



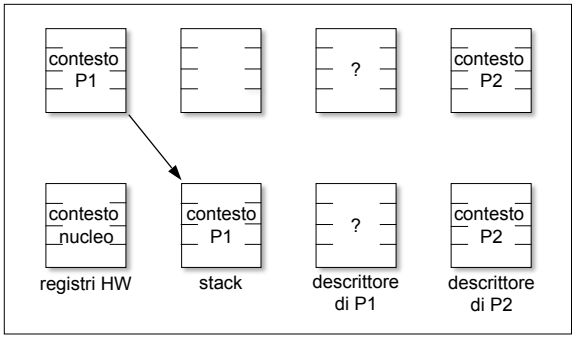
Commutazione di contesto (3)

- La commutazione di contesto dal processo P1 (da esecuzione a attesa) al processo P2 (da pronto a esecuzione) avviene in quattro fasi:
  1. inizialmente il processo P1 è in esecuzione, il processore opera nel contesto di P1, lo stack contiene dati locali di P1, il descrittore di P1 non è significativo, il descrittore di P2 contiene il contesto di P2 salvato quando P2 ha interrotto l'esecuzione



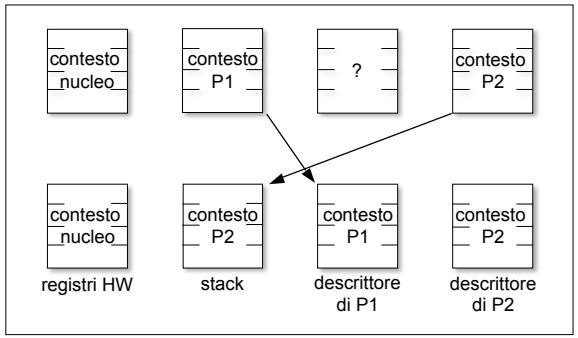
Commutazione di contesto (4)

2. P1 esegue una SVC per richiedere una operazione di I/O. Il suo contesto viene posto in cima allo stack e il processore opera nel contesto del nucleo



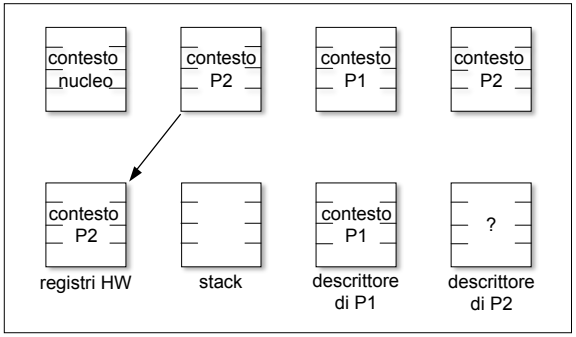
Commutazione di contesto (5)

3. Il nucleo porta P1 in stato di attesa e P2 in stato di esecuzione, salvando il contesto di P1 nel descrittore di P1, e ripristinando dal descrittore di P2 il contesto di P2



Commutazione di contesto (6)

4. Il nucleo termina la SVC eseguendo un ritorno da interruzione che ripristina il contesto del processore con il contenuto dello stack. Il processore opera nel contesto di P2



Strutture dati del sistema operativo

- Mantengono informazioni sullo stato corrente del sistema in termini di processi e risorse
  - tabella dei processi
  - tabella di allocazione di memoria
  - tabella dei dispositivi di I/O
  - tabella dei file aperti

Tabella dei processi

- Identificatore di processo
- Allocazione in memoria (segmenti)
- File utilizzati
- Programma eseguito
- Informazioni di stato
- Informazioni contabili

Tabella di allocazione di memoria

- Allocazione della memoria centrale ai processi
- Allocazione di memoria secondaria ai processi
- Attributi di protezione per l'accesso a zone di memoria condivisa
- Informazioni necessarie per la gestione della memoria virtuale

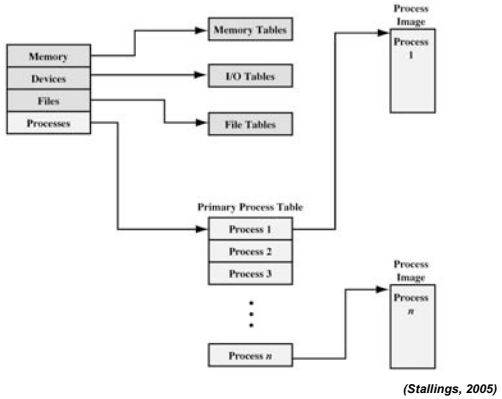
Tabella dei dispositivi di I/O

- Stato dei dispositivi di I/O
  - disponibile
  - occupato
  - assegnato esclusivamente ad un processo
- Stato delle operazioni di I/O
- Informazioni sui buffer utilizzati per il trasferimento dei dati da/verso la periferia

Tabella dei file aperti

- Identificazione dei file
- Locazione sulla memoria secondaria
- Stato corrente di accesso / condivisione / posizione di lettura e scrittura
- Attributi
- L'informazione può essere gestita attraverso il file system

Relazioni tra le strutture dati del S.O.



(Stallings, 2005)

I processi in Unix (1)

- La gestione dei processi in Unix si basa sui concetti di *processo* e di *immagine*
  - il processo è l'entità attiva che esegue un programma (l'immagine); è descritto da un identificatore di processo, da una struttura dati (descrittore), e corrisponde all'insieme di *codice*, *dati utente* e *dati di nucleo*
  - l'immagine è il testo del programma eseguito dal processo; è composta da un'area contenente il codice, e da un'area riservata per i dati del programma eseguito (*dati utente* e stack)
  - il processo è un'entità dinamica, il programma eseguito è un'entità statica

I processi in Unix (2)

- Più processi possono evolvere contemporaneamente, e sono detti *attivi*
- Su macchine dotate di un solo processore, un solo processo tra quelli attivi è in esecuzione in un certo istante
- Quando un processo è in esecuzione la sua immagine deve essere presente in memoria centrale

Creazione di processi in Unix

- La creazione di un processo e la definizione della sua immagine avvengono attraverso un meccanismo combinato
  - duplicazione di un processo esistente (*fork*), che dà origine ad un processo (detto processo figlio) copia del processo esistente (detto processo padre)
  - sostituzione dell'immagine eseguita (*exec*), che permette ad uno dei due processi di evolvere separatamente dall'altro

La funzione fork

```
esito = fork();
```

- Crea una copia del processo che esegue la fork
- L'area dati viene duplicata, l'area codice viene condivisa
- Il processo creato (*figlio*) riceve esito = 0
- Il processo creante (*padre*) riceve esito > 0 e uguale all'identificatore di processo del processo creato
- Se l'operazione fallisce esito < 0

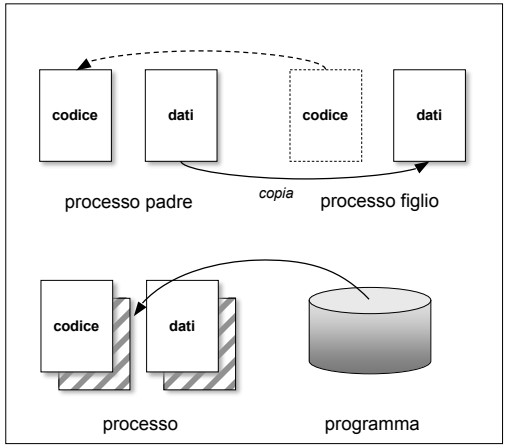
La funzione exec

```
exec(nome file,lista argomenti);
```

- Sostituisce nel processo che la esegue l'immagine con il contenuto del file eseguibile indicato come primo parametro
- L'esecuzione prosegue con il nuovo programma a cui vengono trasmessi gli argomenti specificati
- Esistono più varianti della funzione che differiscono per la struttura dei parametri
  - execl(file, arg1, arg2, …, argN, 0)
  - execv(file, argv)

Le funzioni fork e exec

- fork
- exec

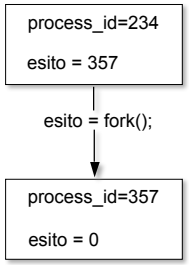


Relazioni tra i processi

- Il processo figlio non condivide memoria con il processo padre (ne condivide il codice)
  - dalla sua creazione in poi i due processi evolvono separatamente eseguendo la stessa immagine
- La creazione avviene per duplicazione completa (logica) del processo padre
  - il processo figlio eredita l'ambiente di lavoro: file aperti, privilegi, directory di lavoro, etc.

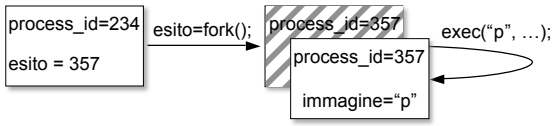
Un esempio di creazione di un processo

```
esito = fork();
if (esito < 0)
 {
   /* la fork ha fallito ... */
 }
else if (esito > 0)
 {
   /* codice del processo padre */
 }
else
 {
   /* codice del processo figlio */
 }
```

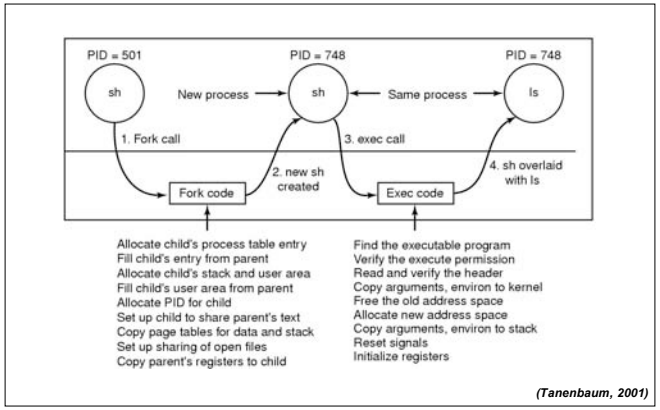


Un esempio di esecuzione di un'immagine

```
esito = fork();      /* crea un processo figlio */
if (esito == 0)      /* se è il figlio */
 { exec("p",...);    /* esegue il programma "p" */
   error(...);       /* ...a meno di errori */
 }
...                          /* qui procede solo il padre */
```



Fork + exec per l'esecuzione di comandi shell



(Tanenbaum, 2001)

Terminazione di processi in Unix

- La terminazione di un processo consiste in una serie di operazioni che lasciano il sistema in stato coerente
  - chiusura dei file aperti
  - rimozione dell'immagine dalla memoria
  - eventuale segnalazione al processo padre
- Per gestire quest'ultimo aspetto Unix impiega due funzioni in modo coordinato
  - terminazione dell'esecuzione di un processo (*exit*)
  - attesa della terminazione di un processo da parte del processo che lo ha creato (*wait*)

Le funzioni exit e wait

```
exit(stato);
```

- Termina l'esecuzione di un processo segnalando al processo che lo ha creato un valore numerico che rappresenta l'esito sintetico (stato) dell'esecuzione

```
process_id=wait(&stato);
```

- Attende la terminazione di un processo figlio; restituisce l'identificativo del processo terminato e il suo stato di terminazione

Un esempio riassuntivo

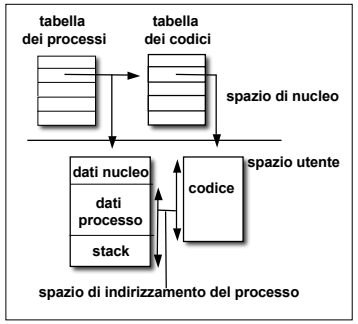
```
esito = fork();       // crea un processo figlio
if (esito < 0)        // creazione OK?
  { error("fork() non eseguita");
    ...
  }
if (esito == 0)       // è il processo figlio ?
  { exec("p",...);    // sì, esegue il programma "p"
    error("exec non eseguita");          // ...a meno di errori
    ...
  }
id = wait(&stato);  // il processo padre attende la fine
if (stato == ...)     // del figlio e ne analizza l'esito
  {
    ...
  }
```

Implementazione dei processi in Unix (1)

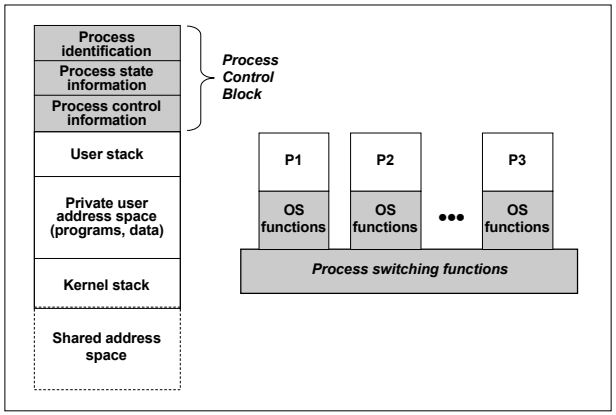
- Ogni processo è diviso in due parti
  - codice: contiene le istruzioni del programma eseguito dal processo
  - dati: contiene i dati su cui il processo opera (variabili del programma), lo stack e un'area riservata per dati necessari al sistema operativo per gestire la commutazione dei processi (area dati di nucleo)
- Area codice e area dati sono allocate separatamente in memoria

Implementazione dei processi in Unix (2)

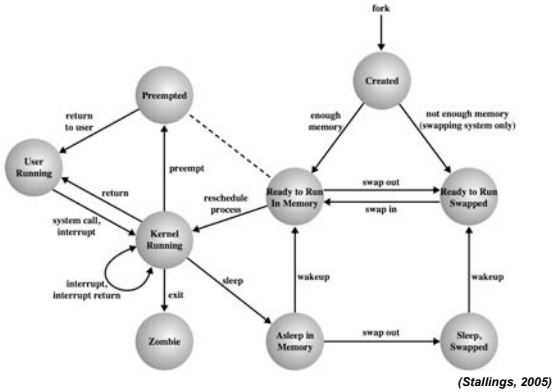
- Il sistema operativo mantiene traccia dei processi in esecuzione e dei relativi programmi attraverso due strutture dati principali
  - tabella dei processi: ogni processo è individuato attraverso un identificatore di processo (numerico)
  - tabella dei codici: ogni programma (in formato eseguibile) è presente una sola volta, e condiviso tra tutti i processi che eseguono quel programma



Esecuzione del S.O. nei processi utente

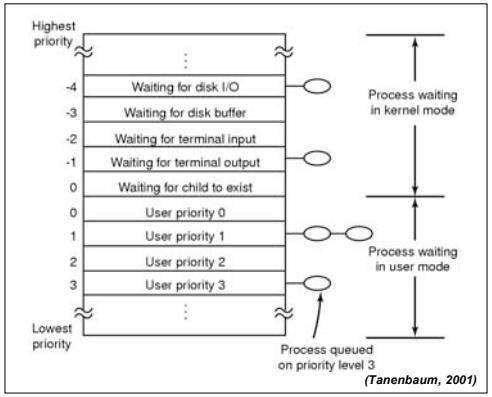


Gli stati dei processi in Unix



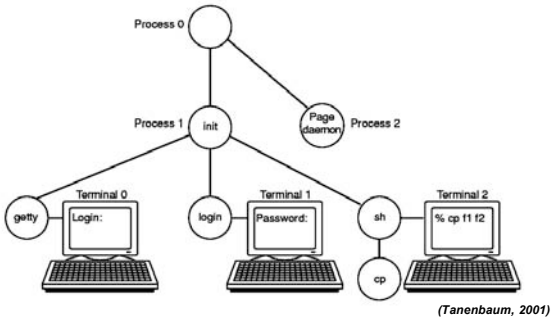
(Stallings, 2005)

Le priorità di attesa dei processi in Unix



(Tanenbaum, 2001)

Il bootstrap di Unix



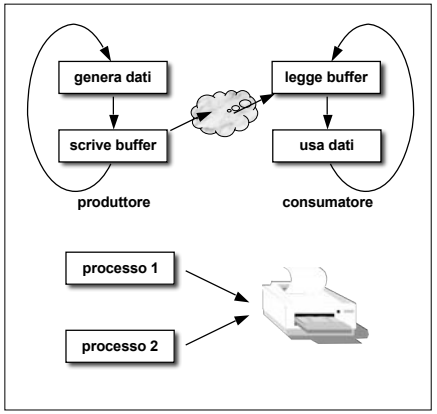
(Tanenbaum, 2001)

Cooperazione e competizione tra processi (1)

- Due o più processi possono interagire fra loro secondo due modalità: *cooperazione* e *competizione*
  - due processi cooperano se sono logicamente connessi e ciascuno ha bisogno dell'altro per operare
  - l'interazione è desiderata e prevista
  - due processi competono se potrebbero evolvere indipendentemente ma entrano in conflitto sulla ripartizione di risorse
  - l'interazione è non desiderata e non prevista

Cooperazione e competizione tra processi (2)

- Cooperazione
- Competizione



Cooperazione e competizione tra processi (3)

- In entrambi i casi occorre predisporre meccanismi di sincronizzazione e comunicazione che permettano ai processi di gestire la cooperazione o la competizione:
  - attraverso l'uso di dati comuni (*memoria condivisa*)
  - i processi condividono parte dei loro dati: le modifiche effettuate da un processo possono essere rilevate da un altro processo
  - attraverso lo *scambio di messaggi*
  - un processo trasmette le informazioni ad un altro processo attraverso operazioni simili alle operazioni di I/O

Proprietà dei processi (1)

- Una coppia (o un insieme) di processi cooperanti
  - può condividere una parte dei dati
  - può sincronizzarsi in momenti selezionati dell'esecuzione
- Ciascun processo evolve anche indipendentemente dagli altri
  - i processi operano (anche) su dati privati
- L'assegnazione della CPU ai processi è regolata dal sistema operativo
  - in base alle politiche di scheduling
  - attraverso meccanismi di commutazione di contesto

Proprietà dei processi (2)

- Se i processi hanno poche interrelazioni il cambiamento di contesto non altera in modo sostanziale le prestazioni complessive
- Se i processi hanno molte interrelazioni e/o condividono molti dati, la realizzazione con processi separati può introdurre un overhead sensibile
  - scheduling
  - sincronizzazione
  - gestione memoria

  } attraverso il kernel

Proprietà dei processi (3)

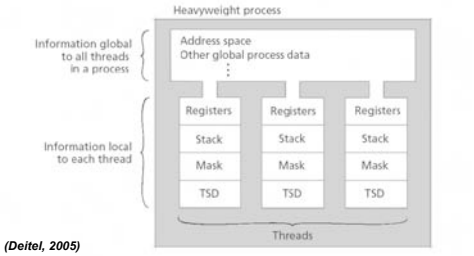
- Il processo è un'unità di allocazione di risorse
  - memoria virtuale per l'immagine del processo
  - controllo su altre risorse esterne (dispositivi I/O, file, …)
- Il processo è un'unità di esecuzione (*dispatching*)
  - identifica un flusso di esecuzione attraverso uno o più programmi
  - l'esecuzione può essere intervallata / sincronizzata con quella di altri processi
  - un processo ha uno stato di esecuzione e alcuni attributi che ne determinano le modalità di esecuzione (es. priorità)

Proprietà dei processi (4)

- Queste due proprietà possono essere gestite in modo indipendente
  - l'unità di esecuzione è identificata dal concetto di thread (lightweight process)
  - l'unità di allocazione delle risorse è identificata dal concetto di processo
- Con i thread si introduce una struttura di esecuzione più articolata, basata su
  - condivisione di risorse
  - differenziazione del flusso di esecuzione all'interno di un unico processo

Thread (1)

- Un thread è una unità di impiego di CPU all'interno di un processo
- Un processo può contenere più thread, ciascuno dei quali evolve in modo logicamente separato dagli altri thread



(Deitel, 2005)

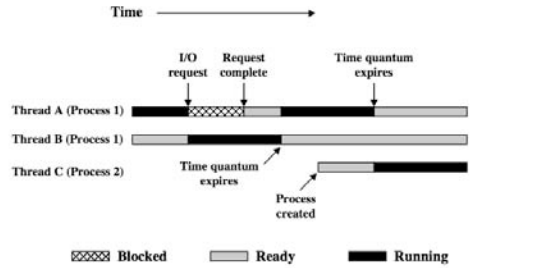
Thread (2)

- Ogni thread è caratterizzato da uno stato di esecuzione
  - program counter
  - un insieme di registri
  - uno stack (dati locali)
- Condivide con gli altri thread dello stesso processo il codice, i dati globali e le risorse dell'ambiente esterno (I/O, file, …)
- Ogni thread viene eseguito in modo logicamente indipendente dagli altri
  - lo spazio di indirizzi è unico
  - i thread possono interagire tra loro (in modo controllato)

Thread (3)

- Un thread può essere attivo, in attesa, pronto, terminato, come un processo
  - non esiste lo stato *suspended* (la memoria è una risorsa del processo)
  - la sospensione di un processo sospende tutti i suoi thread
  - la terminazione di un processo termina tutti i suoi thread
- Un processo è una struttura del sistema operativo, un thread è una sottostruttura del processo (*lightweight process*)
- I thread possono essere implementati a livello utente (librerie) o a livello kernel (system call)

Scheduling dei thread

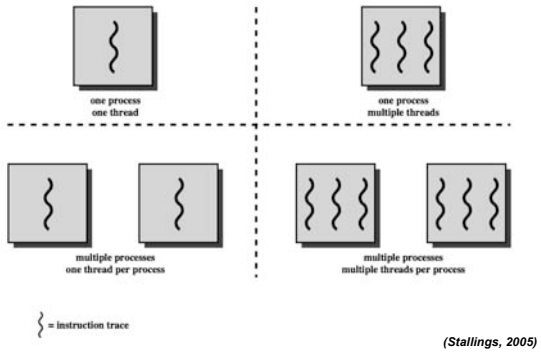


(Stallings, 2005)

Single threading vs. multithreading (1)

- Single threading
  - il sistema operativo non supporta il concetto di thread come entità separata dal processo
  - MS-DOS supporta(va) un solo processo utente con un solo thread di esecuzione
  - UNIX (originale) supporta più processi utente ma solo un thread per processo
- Multithreading
  - il sistema operativo supporta l'esecuzione di più thread all'interno di un processo
  - Windows 2000/XP, Solaris, Linux, supportano thread multipli per ogni processo

Single threading vs. multithreading (2)



(Stallings, 2005)

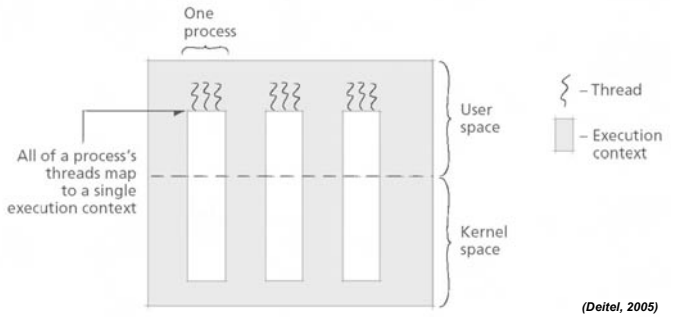
Vantaggi dei thread sui processi

- La gestione dei thread è più veloce rispetto alla gestione dei processi
  - creazione
  - terminazione
  - commutazione di contesto
- I thread possono comunicare attraverso i dati locali invece che attraverso meccanismi di *interprocess communication* (*IPC*)
  - l'accesso ai dati locali deve essere regolamentato
- Si elimina il cambiamento di contesto dovuto all'intervento del sistema operativo
  - solo per i thread di utente

Thread di utente vs. thread di kernel

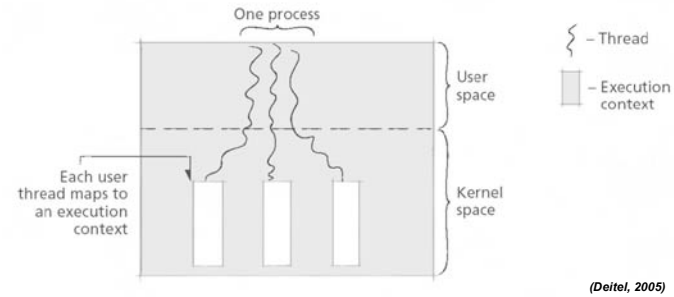
- Thread di utente
  - la gestione è completamente a carico dell'applicazione
  - il kernel non ha cognizione dei thread e non li gestisce
  - realizzato attraverso chiamate a funzioni di libreria
- Thread di kernel
  - il kernel gestisce le informazioni sul contesto dei processi e dei thread
  - lo scheduling delle attività si basa sui thread e non sui processi
  - implementato in Windows 2000/XP e Linux
- Una soluzione mista combina le proprietà di entrambi
  - implementato in Solaris

Thread di utente



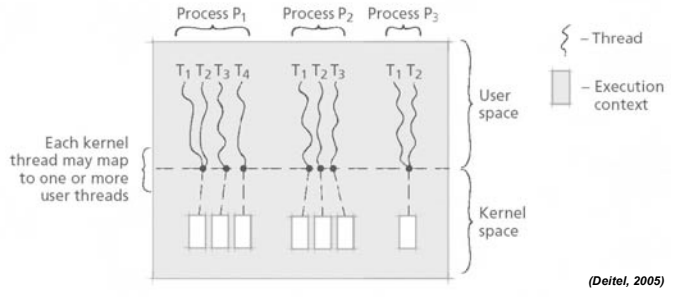
(Deitel, 2005)

Thread di kernel



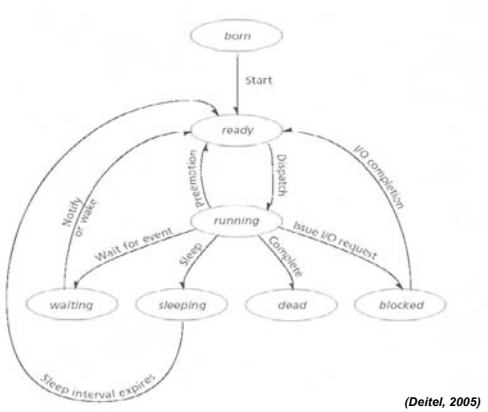
(Deitel, 2005)

Modello misto di thread



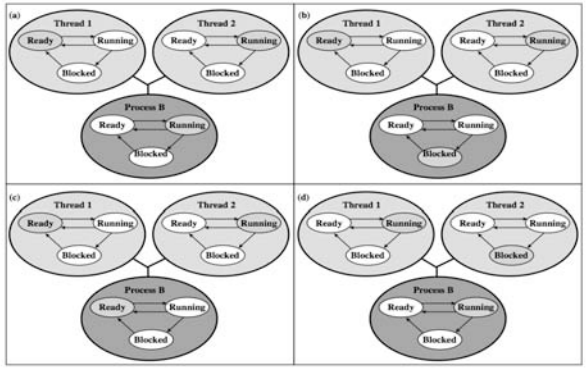
(Deitel, 2005)

Gli stati di un thread



(Deitel, 2005)

Relazioni tra lo stato dei thread e quello del processo



(Stallings, 2005)

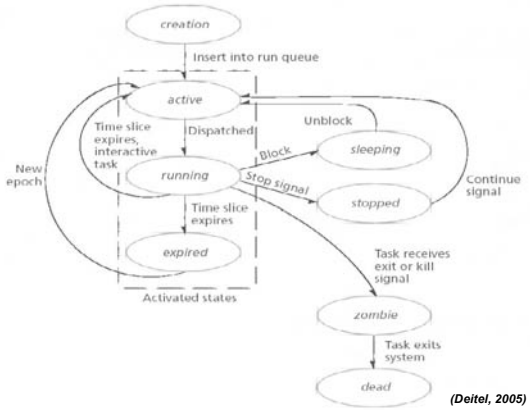
Thread di utente: vantaggi e problemi

- Vantaggi
  - la commutazione tra i thread non richiede l'intervento del kernel
  - lo scheduling dei processi è indipendente da quello dei thread
  - lo scheduling può essere ottimizzato per la specifica applicazione
  - possono essere implementati su qualunque sistema operativo attraverso una libreria
- Problemi
  - la maggior parte delle system call sono bloccanti, il blocco del processo causa il blocco di tutti i suoi thread
  - nei sistemi multiprocessor il kernel non può assegnare due thread a due processori diversi

Thread di sistema: vantaggi e problemi

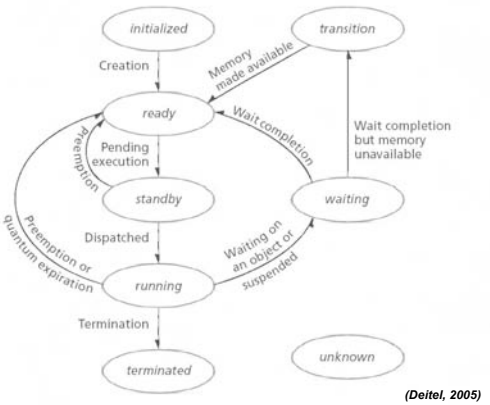
- Vantaggi
  - in un sistema multiprocessor il kernel può assegnare più thread dello stesso processo a processori diversi
  - la sospensione e l'esecuzione delle attività sono eseguite a livello thread
- Problemi
  - la commutazione di thread all'interno di un processo costa quanto la commutazione di processo
  - cade uno dei vantaggi dell'uso dei thread rispetto ai processi

Gli stati dei thread in Linux



(Deitel, 2005)

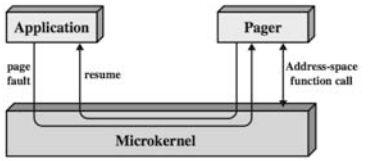
Gli stati dei thread in Windows 2000/XP



(Deitel, 2005)

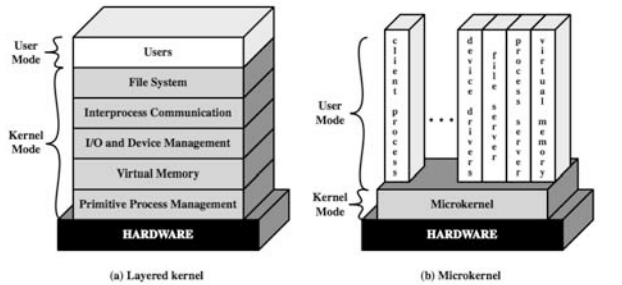
Architetture a microkernel

- Il sistema operativo è composto da un piccolo nucleo che contiene le funzioni fondamentali di gestione dei processi
- Molti servizi tradizionalmente compresi nel sistema operativo sono realizzati come sottosistemi esterni
  - device driver
  - file system
  - gestore della memoria virtuale
  - sistema di windowing e interfaccia utente
  - sistemi per la sicurezza



(Stallings, 2005))

Kernel convenzionale vs. microkernel



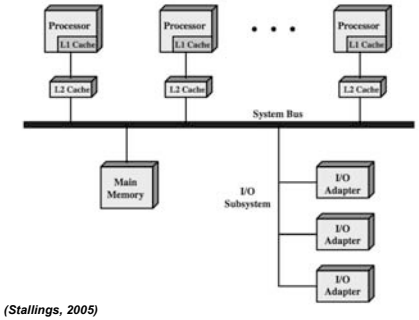
(Stallings, 2005)

Vantaggi di un'architettura a microkernel

- Interfaccia uniforme delle richieste di servizio da parte dei processi
  - tutti i servizi sono forniti attraverso lo scambio di messaggi
- Estendibilità, flessibilità, portabilità
  - è possibile aggiungere, eliminare, riconfigurare nuovi servizi senza toccare il kernel
- Affidabilità
  - progetto modulare, *object oriented design*
  - verificabilità (kernel limitato nelle dimensioni e nelle funzioni)
- Supporto per i sistemi distribuiti
  - lo scambio di messaggi è indipendente dall'organizzazione reciproca di mittente e destinatario

Simmetric multiprocessing (SMP)

- Il kernel può essere eseguito su qualunque processore di un sistema multiprocessor
- Ogni processore gestisce il proprio scheduling nell'ambito dei processi e dei thread disponibili



(Stallings, 2005)

Tipi di processi

- Processi sequenziali
- Processi concorrenti
- Processi in tempo reale

Processi sequenziali

- Sono processi il cui comportamento non è influenzato dalla presenza di eventi esterni
  - il loro comportamento è indipendente dalla velocità di esecuzione
  - il processo può essere rallentato arbitrariamente senza alterazione nei suoi risultati
  - se un processo viene rieseguito con lo stesso programma e con gli stessi dati produce gli stessi risultati

Processi concorrenti

- Sono processi il cui comportamento è influenzato dalla contemporanea presenza di altri processi
  - il comportamento è indipendente dalla velocità di esecuzione, purché esista la possibilità di sincronizzarsi su punti specifici dell'esecuzione (es. produttore-consumatore)
  - se una coppia di processi viene rieseguita con gli stessi programmi, gli stessi dati, e gli stessi meccanismi di sincronizzazione, produce gli stessi risultati

Processi in tempo reale

- Sono processi il cui comportamento è influenzato da eventi esterni la cui temporizzazione non è prevedibile né modificabile
  - il comportamento dipende dalla velocità di esecuzione, perché dipende dalla risposta agli eventi esterni
  - la riesecuzione con lo stesso programma e con gli stessi dati può dare risultati diversi perché gli eventi esterni non si ripetono in modo identico