Sistemi Operativi T

Processi e Thread

Concetto di processo

Il processo è un programma in esecuzione

- È l'**unità di esecuzione** all'interno del SO
- Solitamente, esecuzione **sequenziale** (istruzioni vengono eseguite in sequenza, secondo l'ordine specificato nel testo del programma)
- SO multiprogrammato consente ***l'esecuzione concorrente di più processi***

D'ora in poi faremo implicitamente riferimento sempre al caso di SO multiprogrammati

Concetto di processo

Programma = entità passiva
Processo = entità attiva

Il processo è rappresentato da:

- **codice** (*text*) del programma eseguito
- **dati**: variabili globali
- **program counter**
- alcuni **registri** di CPU
- **stack**: parametri, variabili locali a funzioni/procedure

Inoltre, a un processo possono essere associate delle ***risorse di SO***. Ad esempio:

- file aperti
- connessioni di rete
- altri dispositivi di I/O in uso
- ...

Processo = {PC, registri, stack, text, dati, risorse}

Stati di un processo

Un processo, durante la sua esistenza può trovarsi in vari **stati**:

- Init: ***stato transitorio*** durante il quale il processo viene caricato in memoria e SO inizializza i dati che lo rappresentano
- Ready: processo è ***pronto per acquisire la CPU***
- Running: processo ***sta utilizzando la CPU***
- Waiting: processo è ***sospeso in attesa di un evento***
- Terminated: ***stato transitorio*** relativo alla fase di terminazione e deallocazione del processo dalla memoria

Stati di un processo

Transizioni di stato:

Stati di un processo

Transizioni di stato:

Stati di un processo

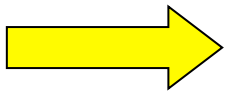
In un sistema ***monoprocessore e multiprogrammato***:

- ***un solo processo*** (al massimo) si trova nello ***stato running***
- più processi possono trovarsi negli ***stati ready e waiting***

  necessità di ***strutture dati per mantenere in memoria le informazioni su processi in attesa***

  ✓ di acquisire la CPU (ready)

  ✓ di eventi (waiting)



Descrittore di processo

Rappresentazione dei processi

Come vengono rappresentati i processi in SO?

- Ad ogni processo viene associata una struttura dati (descrittore): **Process Control Block (PCB)**
- **PCB** contiene tutte le ***informazioni relative al processo***:
  - ✓ Stato del processo;
  - ✓ Contenuto dei registri di CPU (PC, SP, IR, accumulatori, ...)
  - ✓ Informazioni di scheduling (priorità, puntatori alle code, ...)
  - ✓ Informazioni per gestore di memoria (registri base, limite, ...)
  - ✓ Informazioni relative all'I/O (risorse allocate, file aperti, ...)
  - ✓ Informazioni di accounting (tempo di CPU utilizzato, ...)
  - ✓ ...

Process Control Block

Il sistema operativo gestisce i PCB di tutti i processi, organizzandoli in opportune strutture dati
(ad esempio *code* di processi)

Scheduling dei processi

È l'attività mediante la quale SO effettua delle scelte tra i processi, riguardo a:

- ***caricamento in memoria centrale***
- ***assegnazione della CPU***

In generale, SO compie **tre diverse attività di scheduling:**

- scheduling **a breve termine** (o di CPU)
- scheduling **a medio termine** (o swapping)
- scheduling **a lungo termine**

Scheduler a lungo termine

Lo scheduler a lungo termine è quella componente del SO che **seleziona i programmi** da eseguire **dalla memoria secondaria** per caricarli in memoria centrale (creando i corrispondenti processi):

- controlla il ***grado di multiprogrammazione*** (numero di processi contemporaneamente presenti nel sistema)
- è una componente importante dei sistemi **batch multiprogrammati**
- nei sistemi **time sharing non è presente:**

  Interattività: spesso è l'utente che stabilisce direttamente il grado di multiprogrammazione => ***scheduler a lungo termine non è presente***

Scheduler a medio termine (swapper)

Nei sistemi operativi multiprogrammati:

- quantità di ***memoria fisica può essere minore*** della somma delle dimensioni degli ***spazi logici*** di indirizzi da allocare a ciascun processo
- ***grado di multiprogrammazione non*** è, in generale, ***vincolato*** dalle esigenze di spazio dei processi

Swapping: ***trasferimento temporaneo in memoria*** secondaria di processi (o di parti di processi), in modo da consentire il caricamento di altri processi

Scheduler a breve termine (o di CPU)

È quella parte del SO che si occupa della selezione dei processi a cui assegnare la CPU

Nei sistemi time sharing, allo scadere di ogni quanto di tempo, il SO:

- decide ***a quale processo*** assegnare la CPU
  (scheduling di CPU)
- effettua il ***cambio di contesto (context switch)***

Cambio di contesto

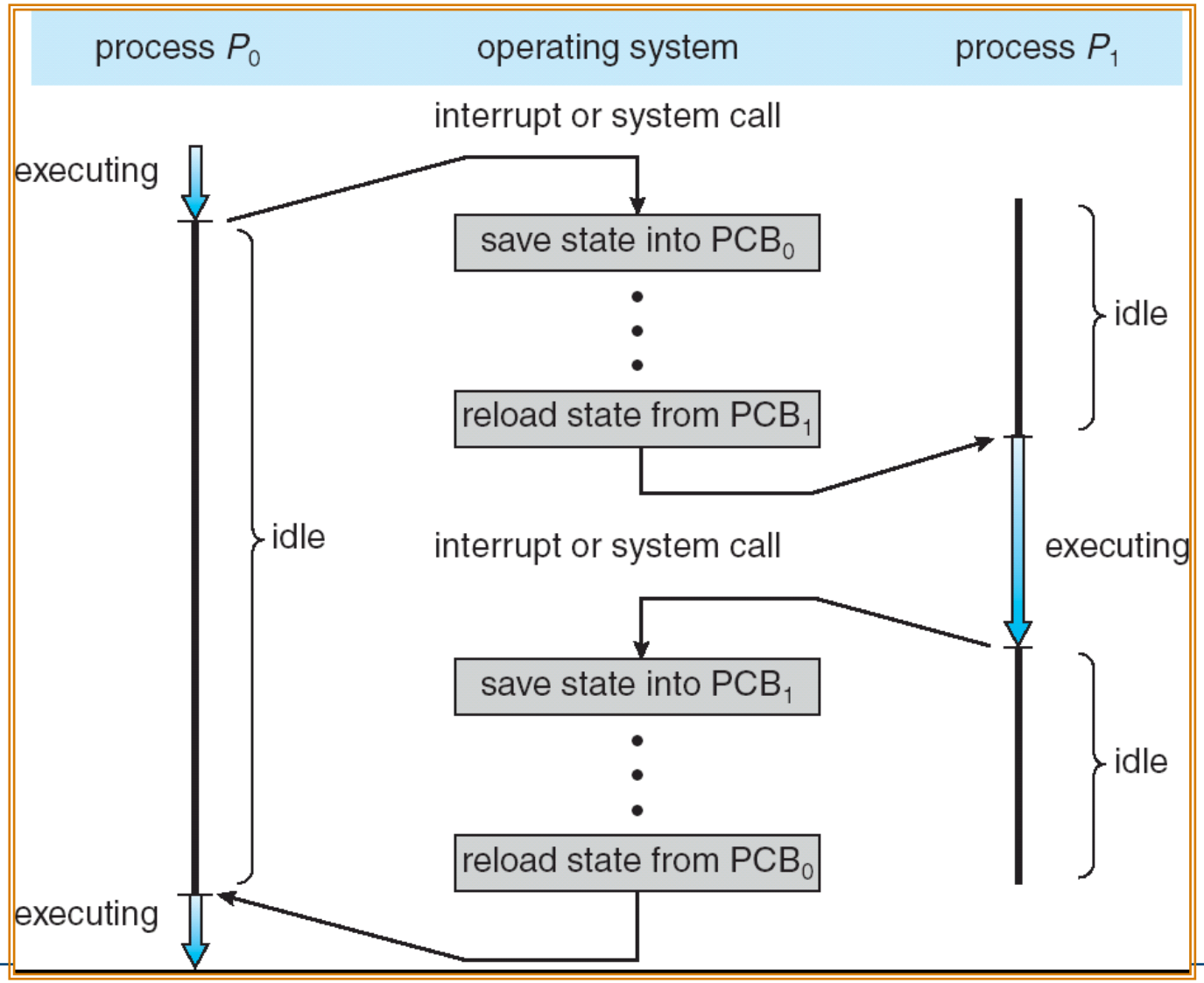
È la fase in cui ***l'uso della CPU viene commutato da un processo ad un altro***

Quando avviene un **cambio di contesto** tra un processo P_i ad un processo P_{i+1} (ovvero, P_i cede l'uso della CPU a P_{i+1}):

- **Salvataggio dello *stato* di P_i**: SO copia PC, registri, ... del processo ***deschedulato*** P_i nel suo PCB
- **Ripristino dello *stato* di P_{i+1}**: SO trasferisce i dati del processo P_{i+1} dal suo PCB nei registri di CPU, che può così riprendere l'esecuzione

➔ Il passaggio da un processo al successivo può richiedere ***onerosi trasferimenti da/verso la memoria secondaria***, per allocare/deallocare gli spazi di indirizzi dei processi (vedi gestione della memoria)

Cambio di contesto



Scheduler a breve termine (o di CPU)

Lo scheduler a breve termine gestisce

- la **coda dei processi pronti:** contiene i PCB dei processi che si trovano in stato *Ready*

Altre strutture dati necessarie

- **code di waiting** (una per ogni tipo di attesa: dispositivi I/O, timer, ...): ***ognuna di esse contiene i PCB dei processi waiting*** in attesa di un evento del tipo associato alla coda

Scheduler

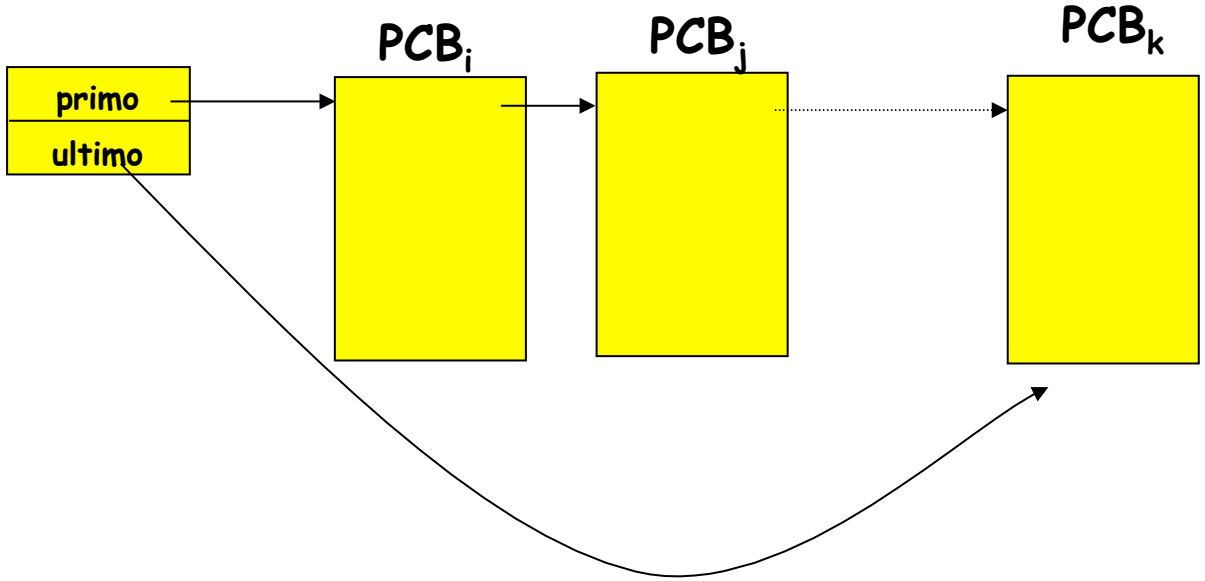
- ***Scheduler short-term*** scheduler viene invocato con ***alta frequenza (ms)*** ⇒ deve essere molto efficiente
- ***Scheduler medium-term*** viene invocato a ***minore frequenza*** (sec-min) ⇒ può essere anche più lento

Scelte ottimali di scheduling dipendono dalla ***tipologia di processi***:

- **processi I/O-bound** – maggior parte del tempo in operazioni I/O, ***molti burst brevi di CPU***
- **processi CPU-bound** – maggior parte del tempo in uso CPU, ***pochi burst di CPU*** tipicamente ***molto lunghi***

Code di Scheduling

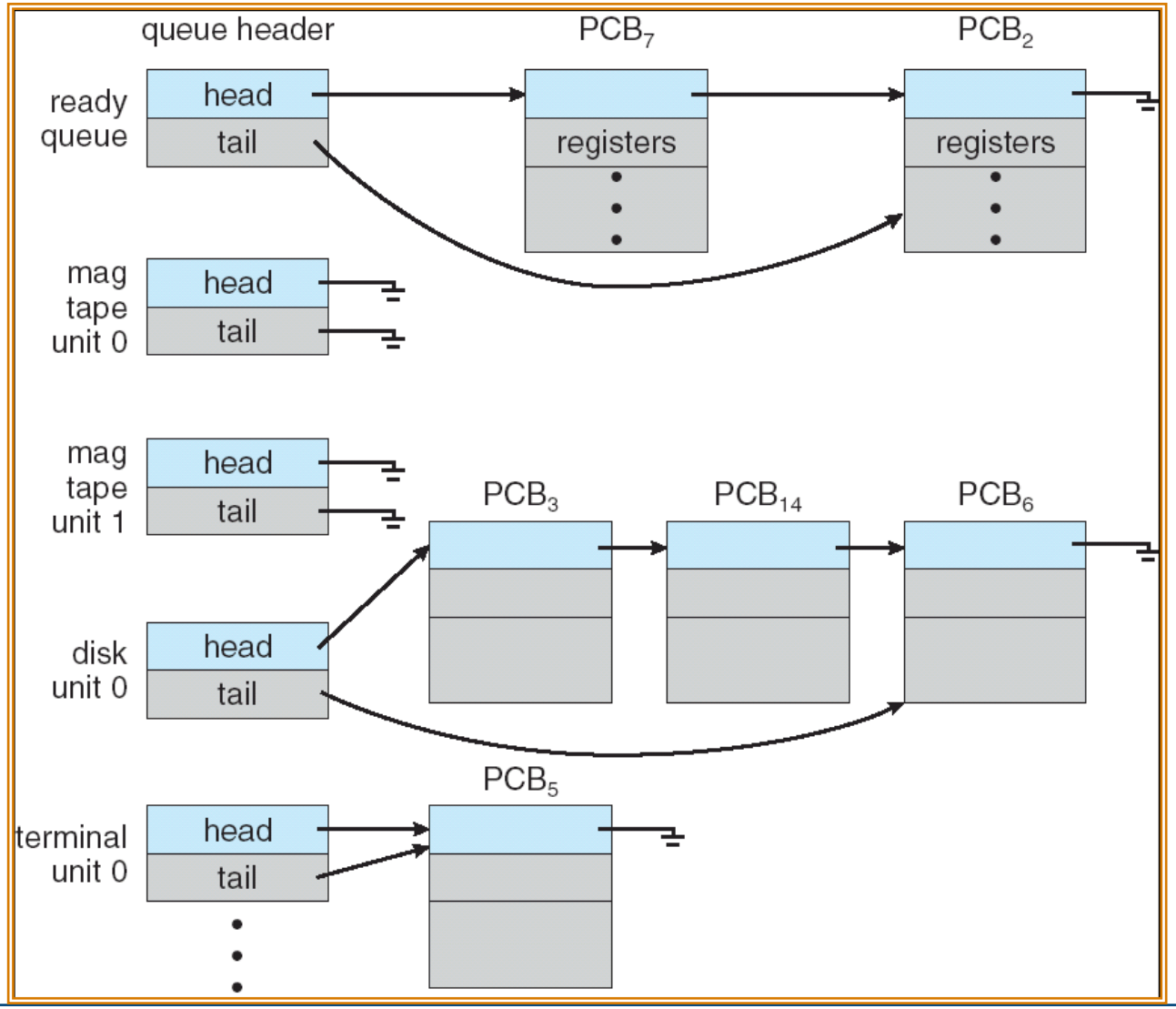
Coda dei processi pronti (*ready queue*):



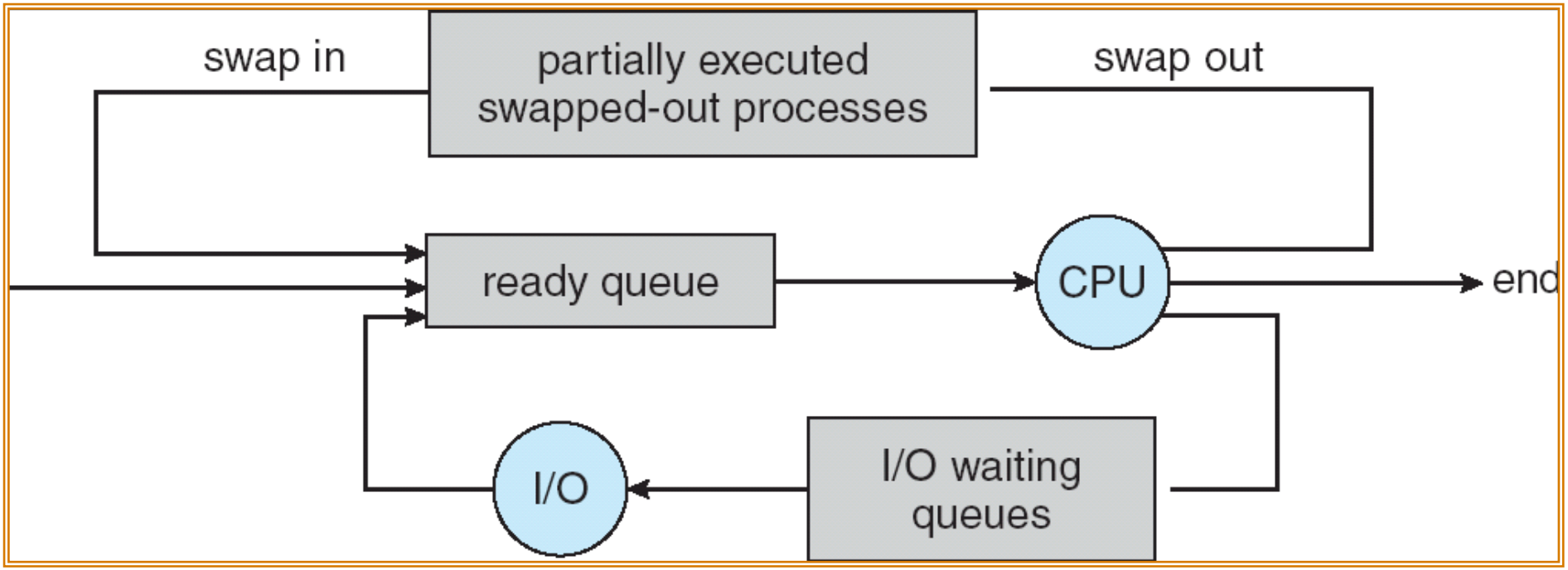
- strategia di gestione della ***ready queue*** dipende dalle ***politiche (algoritmi) di scheduling*** adottate dal SO

  (parleremo di algoritmi di scheduling nelle prossime lezioni)

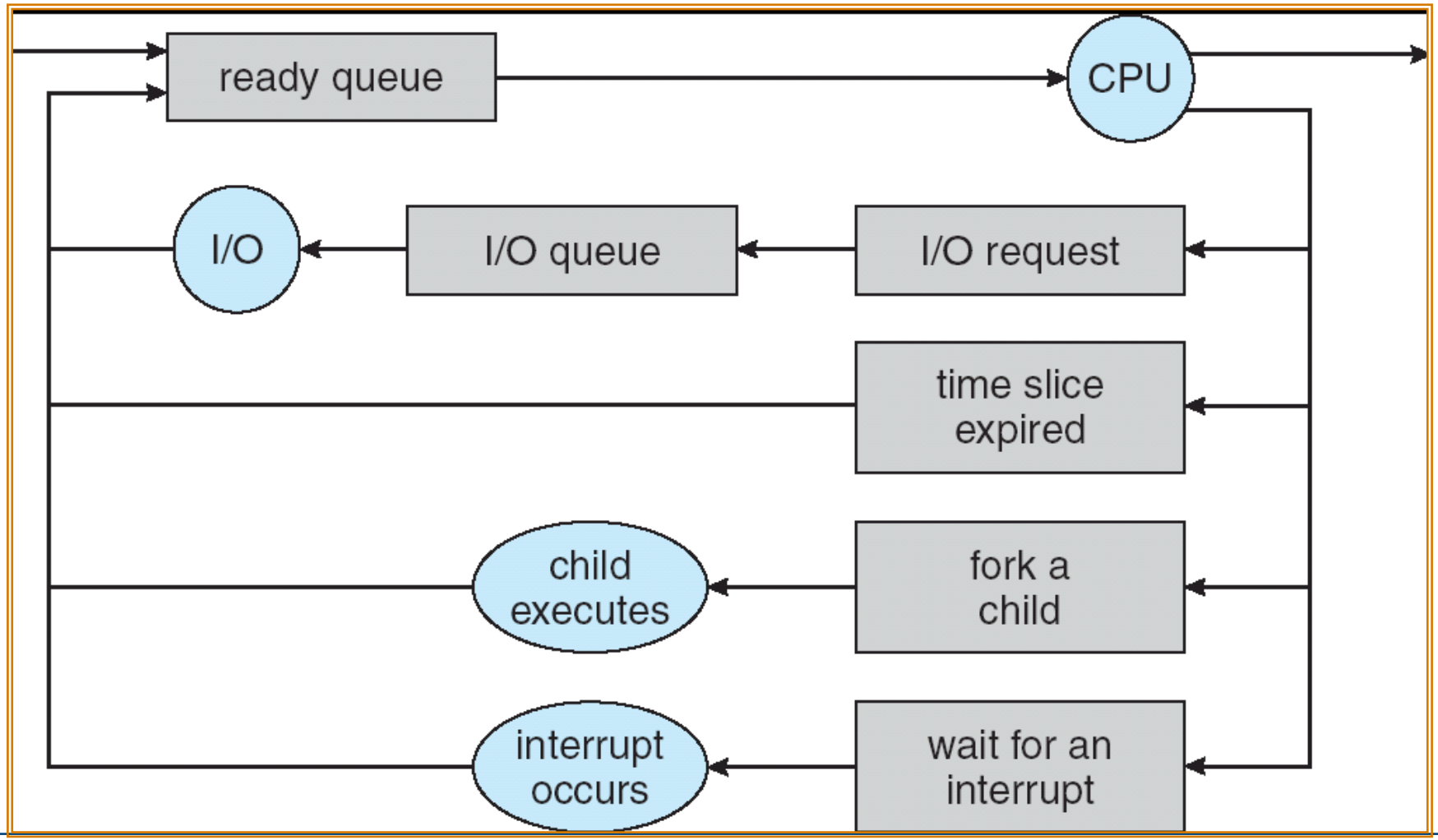
Ready queue e altre code per dispositivi I/O



Short-term + medium-term scheduling



Short-term scheduling



Scheduling e cambio di contesto

Operazioni di scheduling determinano un costo computazionale aggiuntivo che dipende essenzialmente da:

- **frequenza** di cambio di contesto
- dimensione **PCB**
- costo dei trasferimenti da/verso la memoria

➔esistono SO che prevedono **processi** ***leggeri*** (***thread***) che hanno la proprietà di condividere codice e dati con altri processi:

- **dimensione PCB ridotta**
- **riduzione overhead**

Operazioni sui processi

Ogni SO multiprogrammato prevede dei meccanismi per la gestione dei processi

Meccanismi necessari:

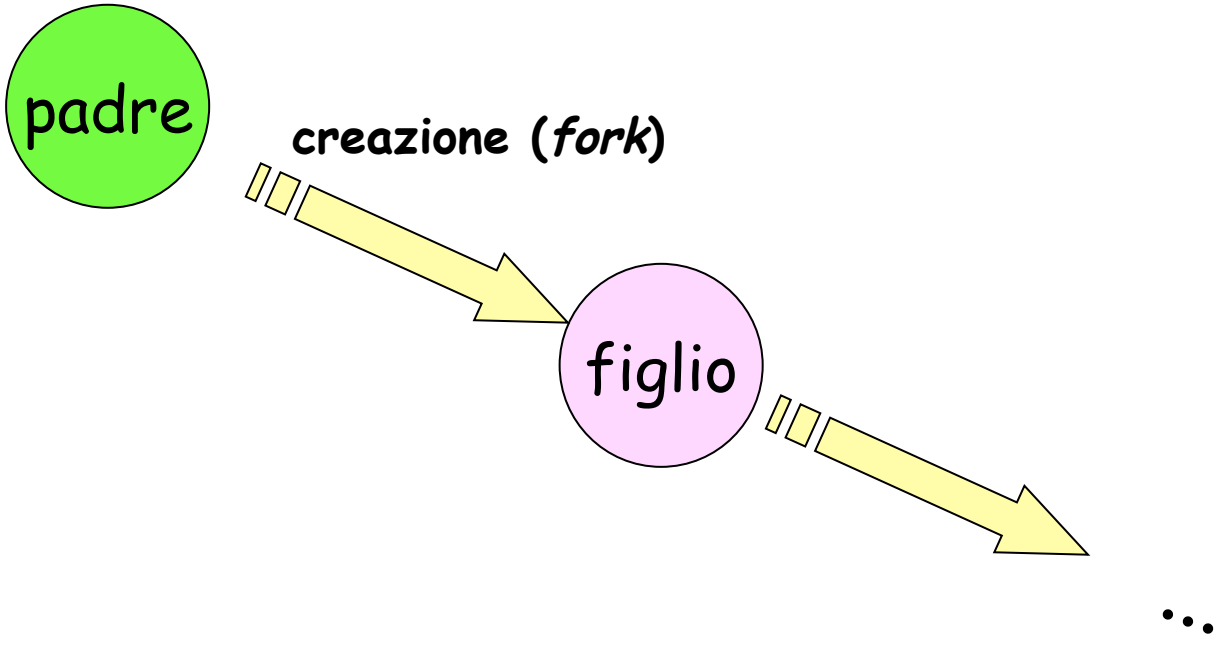
- creazione
- terminazione
- interazione tra processi

Sono operazioni privilegiate (esecuzione in modo kernel)

➔ definizione di **system call**

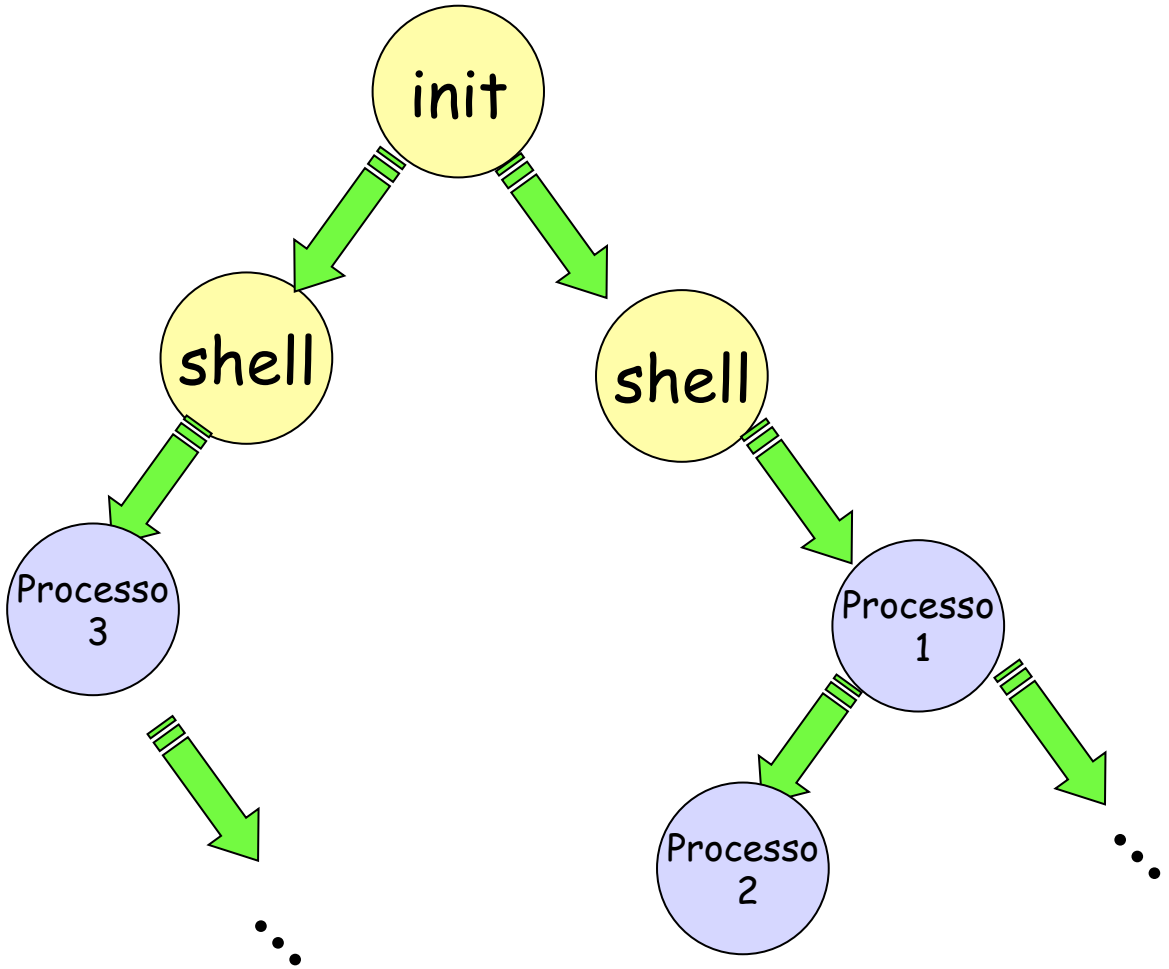
Creazione di processi

- Un processo (*padre*) può richiedere la creazione di un nuovo processo (*figlio*)



- È possibile realizzare **gerarchie** di processi

Gerarchie di processi (es. UNIX)



Relazione padre-figlio

Aspetti caratteristici:

- **concorrenza**
  - padre e figlio procedono in ***parallelo*** (es. UNIX), oppure
  - il padre ***si sospende*** in attesa della terminazione del figlio
- **condivisione di risorse**
  - le risorse del padre (ad esempio, i file aperti) sono ***condivise*** con i figli (es. UNIX), oppure
  - il figlio utilizza risorse soltanto se esplicitamente richieste da se stesso
- **spazio degli indirizzi**
  - ***duplicato***: lo spazio degli indirizzi del figlio è una copia di quello del padre (es. fork() in UNIX), oppure
  - ***differenziato***: spazi degli indirizzi di padre e figlio con codice e dati diversi (es. VMS, stesso processo dopo exec() in UNIX)

Terminazione

Ogni processo:

- è **figlio** di un altro processo
- può essere a sua volta **padre** di processi

SO deve mantenere le informazioni relative alle relazioni di *parentela*

-> nel descrittore: riferimento al padre

Se un processo termina:

- il padre può rilevare il suo ***stato di terminazione***
- tutti i figli terminano

Processi leggeri (thread)

Un thread (o processo leggero) è un'unità di esecuzione che **condivide codice e dati** con altri thread ad esso associati

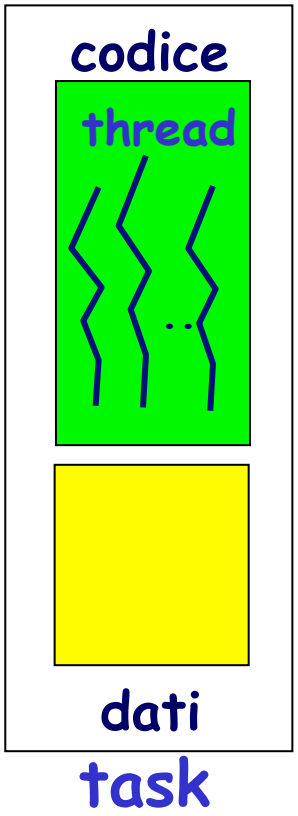
Task = insieme di thread che riferiscono lo stesso codice e gli stessi dati

- **codice e dati non sono caratteristiche del singolo thread, ma del task al quale appartengono**

Thread = {PC, registri, stack, ...}

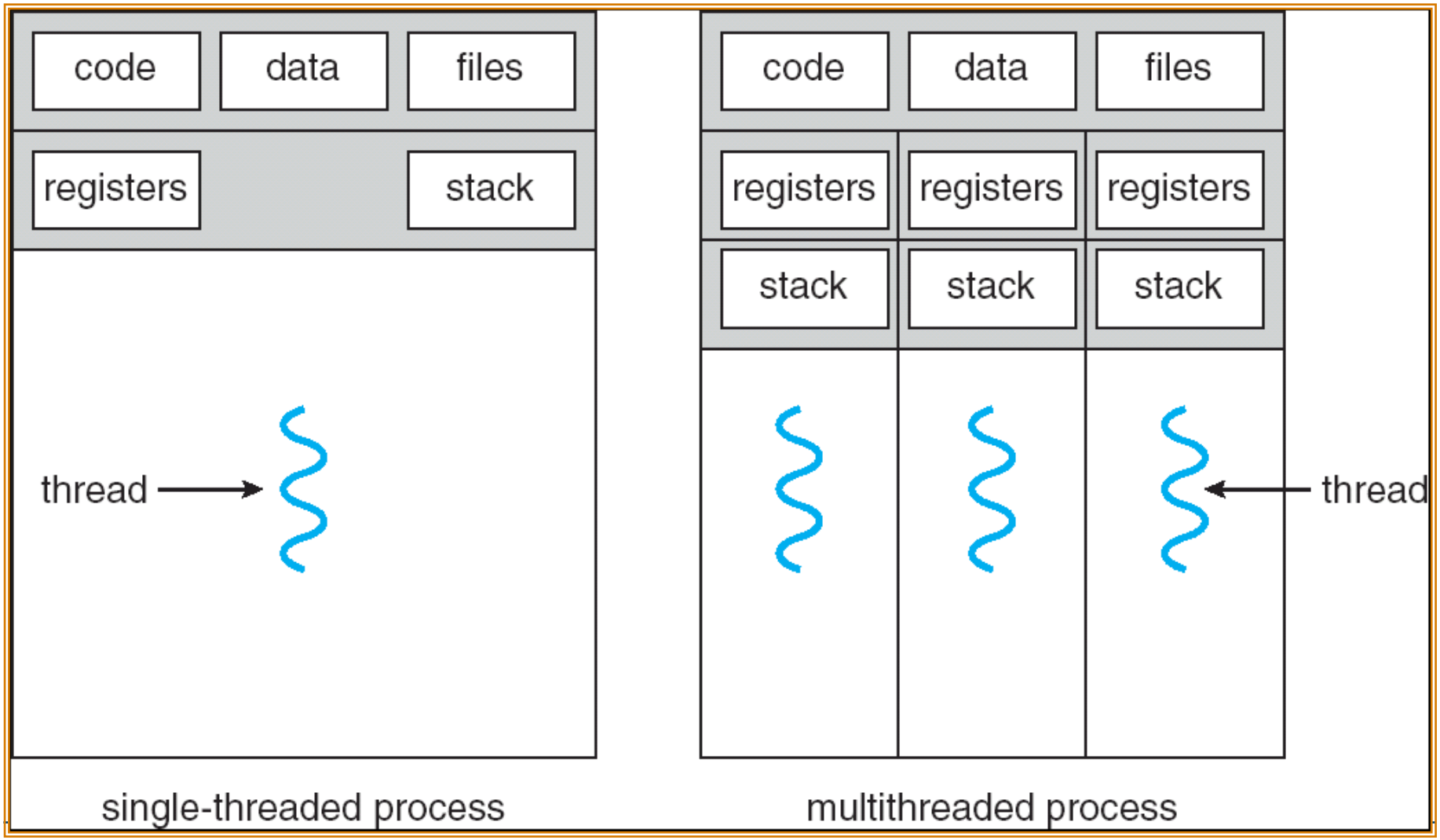
Task = {thread1, thread2, ..., threadN, text, dati}

Processi leggeri (thread)



- **Processo pesante** equivale a un task con un solo thread

Processi single- o multi-threaded



Vantaggi dei thread

- **Condivisione di memoria**: a differenza dei processi (*pesanti*), ***un thread può condividere variabili con altri*** (appartenenti allo stesso task)
- **Minor costo di context switch**: PCB di thread non contiene alcuna informazione relativa a codice e dati

  -> il cambio di contesto fra thread dello stesso task ha un costo notevolmente inferiore al caso dei processi pesanti

- **Minor protezione**: thread appartenenti allo stesso task ***possono modificare dati gestiti da altri thread***

Realizzazione di thread

Alcuni SO offrono anche l'implementazione del concetto di thread (es. MSWinXP, **Linux**, Solaris)

Realizzazione

A livello kernel (MSWinXP, Linux, Solaris, MacOSX):

- SO ***gestisce direttamente i cambi di contesto***
  - tra thread dello stesso task (trasferimento di registri)
  - tra task
- SO fornisce ***strumenti per la sincronizzazione*** nell'accesso di thread a variabili comuni

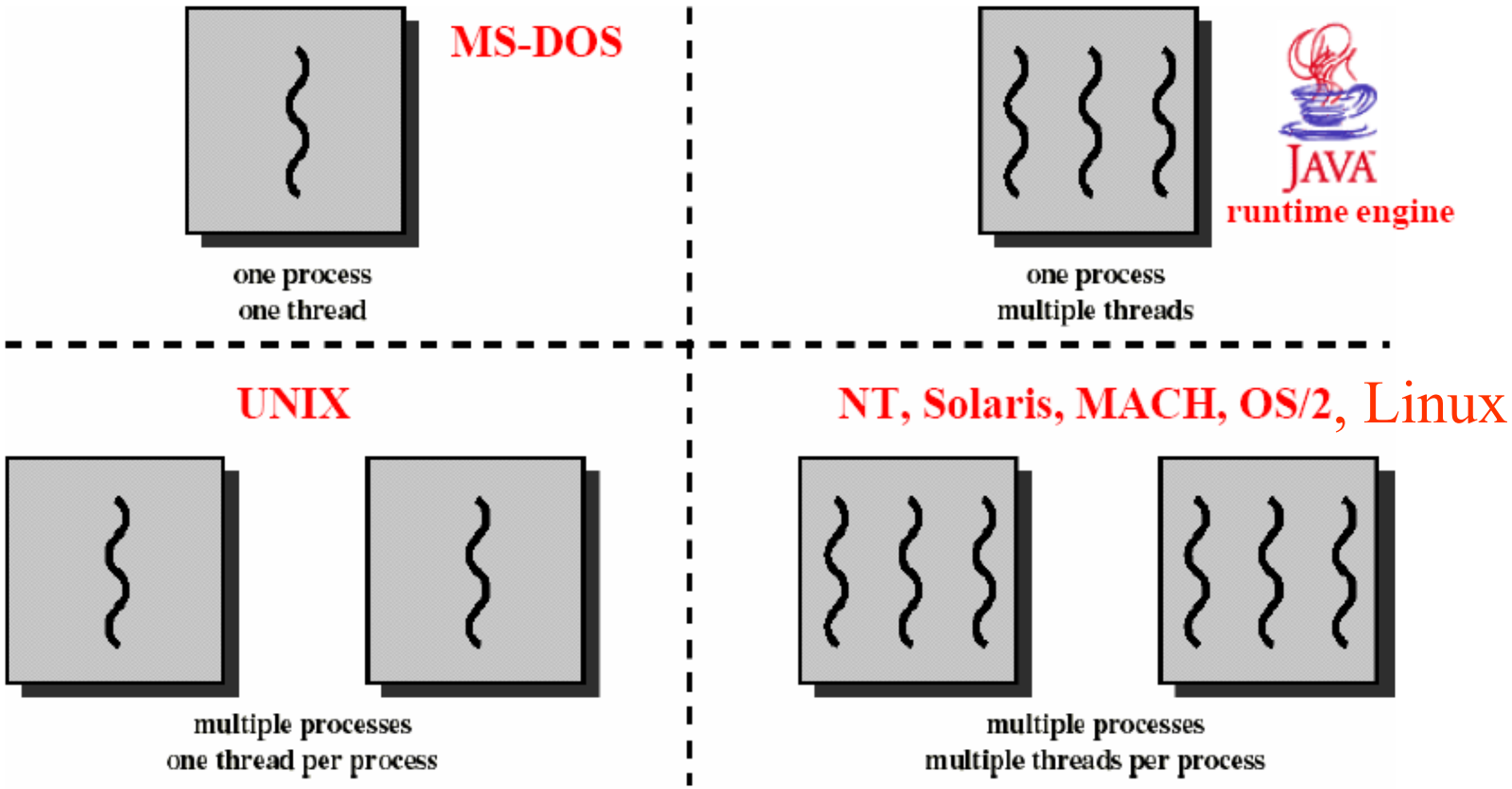
Realizzazione di thread

Realizzazione

- **A livello utente** (es. Andrew - Carnegie Mellon, POSIX pthread, vecchie versioni MSWin, Java thread):
  - il passaggio da un thread al successivo (nello stesso task) ***non richiede interruzioni al SO*** (**maggior rapidità**)
  - SO ***vede processi pesanti***: **minore efficienza**
    - es. sospensione di un thread
    - Cambio di contesto tra thread di task diversi
- **Soluzioni miste**
  - thread realizzati a ***entrambi i livelli***

Soluzioni adottate

- **MS-DOS**: un solo processo utente ed un solo thread.
- **UNIX**: più processi utente ciascuno con un solo thread.
- **Supporto run time di Java**: un solo processo, più thread.
- **Linux, Windows NT, Solaris**: più processi utente ciascuno con più thread.


MS-DOS
one process
one thread
JAVA
runtime engine
one process
multiple threads
UNIX
multiple processes
one thread per process
NT, Solaris, MACH, OS/2, Linux
multiple processes
multiple threads per process

Interazione tra processi

I processi, pesanti o leggeri, *possono* interagire

Classificazione

- **processi indipendenti**: due processi P1 e P2 sono indipendenti se l'esecuzione di P1 non è influenzata da P2, e viceversa
- **processi interagenti**: P1 e P2 sono interagenti se l'esecuzione di P1 è influenzata dall'esecuzione di P2, e/o viceversa

Processi interagenti

Tipi di interazione:

- **Cooperazione:** interazione prevedibile e desiderata, insita nella logica del programma concorrente. I processi cooperanti collaborano per il raggiungmento di un fine comune

- **Competizione:** interazione prevedibile ma "non desiderata" tra processi che interagiscono per sincronizzarsi nell'accesso a risorse comuni

- **Interferenza:** interazione non prevista e non desiderata, potenzialmente deleteria tra processi

Processi interagenti

Supporto all'interazione

L'interazione può avvenire mediante

- **memoria condivisa** (modello ad *ambiente globale*): SO consente ai processi (***thread***) di condividere variabili; l'interazione avviene tramite l'accesso a ***variabili condivise***
- **scambio di messaggi** (modello ad *ambiente locale*): i processi non condividono variabili e interagiscono mediante meccanismi di trasmissione/ricezione di messaggi; SO prevede dei meccanismi a supporto dello ***scambio di messaggi***

Processi interagenti

Aspetti

- ***concorrenza -> velocità***
- suddivisione dei compiti tra processi -> ***modularità***
- ***condivisione di informazioni***
  - assenza di replicazione: ogni processo accede alle stesse istanze di dati
  - necessità di ***sincronizzare i processi*** nell'accesso a dati condivisi

Competizione: mutua esclusione

Esempio

- Due thread P1 e P2 hanno accesso ad una struttura condivisa *organizzata a* **pila** rispettivamente per inserire e prelevare dati.
- La struttura dati è rappresentata da un *vettore* ***stack*** i cui elementi costituiscono i singoli dati e da una *variabile* ***top*** che indica la posizione dell'ultimo elemento contenuto nella pila.
- I thread utilizzano le operazioni ***Inserimento*** *e* ***Prelievo*** per depositare e prelevare i dati dalla pila.

```
typedef ... item;
item stack[N];
int top=-1;

void Inserimento(item y)
{    top++;
                          stack
[top]=y;
}

item Prelievo()
{    item x;
     x= stack[top];
     top--;
     return x;
}
```

- L'esecuzione contemporanea di queste operazioni da parte dei processi può portare ad un uso scorretto della risorsa.

- Consideriamo questa sequenza di esecuzione:

T0:	`top++;`	(P1)
T1:	`x=stack[top];`	(P2)
T2:	`top--;`	(P2)
T3:	`stack[top]=y;`	(P1)

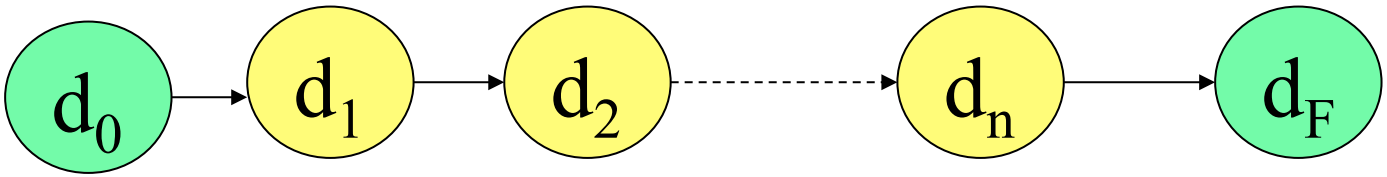
- Viene assegnato a x un valore *non definito* e l'ultimo valore valido contenuto nella pila *viene cancellato* dal nuovo valore di y.
- Potremmo individuare situazioni analoghe, nel caso di esecuzione contemporanea di una qualunque delle due operazioni da parte dei due processi.

- Il problema sarebbe risolto se le due operazioni di Inserimento e Prelievo fossero eseguite sempre in mutua esclusione (istruzioni indivisibili).

Istruzioni Indivisibili

- Data un'istruzione I(d), che opera su un dato d, essa e` indivisibile (o atomica) , se, durante la sua esecuzione da parte di un processo P, il dato d non e` accessibile ad altri processi.
- I(d), a partire da un valore iniziale d_0 , puo` operare delle trasformazioni sullo stato di d, fino a giungere allo stato finale d_F;

➔ poiche` I(d) e` indivisibile, gli stati intermedi non possono essere rilevati da altri processi concorrenti.



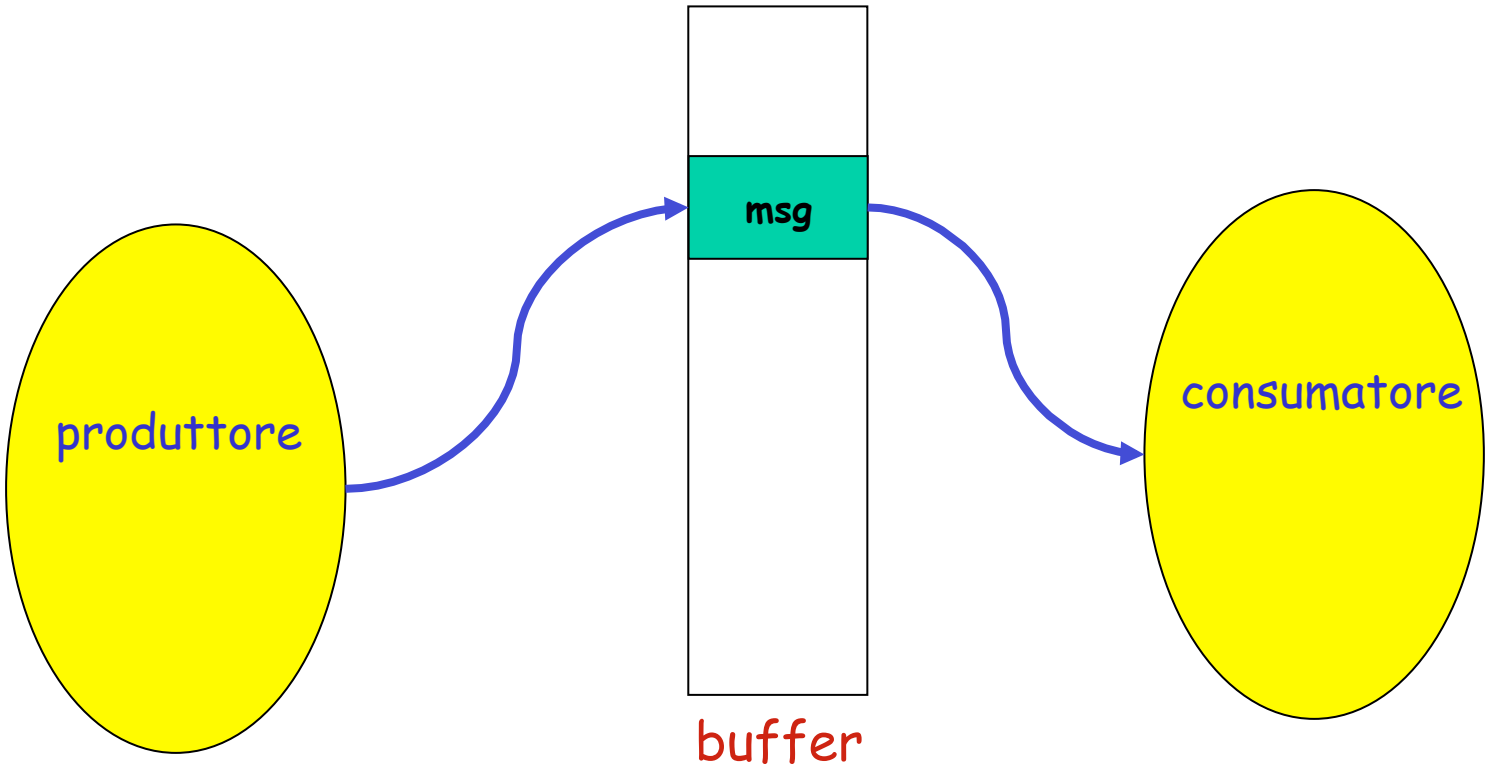
Processi cooperanti

Esempio: produttore & consumatore

Due processi accedono a un buffer condiviso di dimensione limitata:

- un processo svolge il ***ruolo di produttore*** di informazioni che verranno prelevate dall'altro processo (***consumatore***)
- ***buffer*** rappresenta un ***deposito di informazioni condiviso***

Produttore & consumatore



Produttore & consumatore

Necessità di sincronizzare i processi:

- quando il **buffer è vuoto** (il consumatore ***NON può prelevare messaggi***)
- quando il **buffer è pieno** (il produttore ***NON può depositare messaggi***)

Produttore & consumatore

```
Processo produttore:
....
shared msg Buff [DIM];
main()
{ msg M;
   do
   { produco(&M);
     inserisco(M, Buff);
   } while(!fine);
}
```

```
Processo consumatore:
....
shared msg Buff [DIM];
main()
{ msg M;
   do
   { prelievo(&M, Buff);
     consumo(M);
   } while(!fine);
}
```

Problema: che cosa succede se

- il buffer è pieno?
- il buffer è vuoto?

Produttore & consumatore

Necessità di sincronizzare i processi

Aggiungiamo due variabili logiche condivise:

- **buff_vuoto**: se uguale a true, indica che il buffer non contiene messaggi (viene settata dalla funzione di prelievo quando ***l'unico messaggio presente nel buffer viene estratto***)

- **buff_pieno**: se uguale a true, indica che il buffer non può accogliere nuovi messaggi, perché pieno (viene settata dalla funzione di inserimento quando ***l'ultima locazione libera del buffer viene riempita***)

Produttore & consumatore: buffer dim=1

```
Processo produttore:
....
shared int buff_pieno=0;
shared int buff_vuoto=1;
shared msg Buff [DIM];
main()
{ msg M;
   do
   { produco(&M);
     while (buffer_pieno);
     buffer_pieno=1;
     inserisco(M, Buff);
     buffer_vuoto=0;
   } while(!fine);
}
```

```
Processo consumatore:
....
shared int buff_pieno=0;
shared int buff_vuoto=1;
shared msg Buff [DIM];
main()
{ msg
   do
   { while (buffer_vuoto);
     buffer_vuoto=1;
     prelievo(M, Buff);
     buffer_pieno=0;
     consumo(&M);
   } while(!fine);
}
```

Quali problemi?
Come risolverli?