Processi e Thread

Il Concetto di Processo

Il processo è un programma in esecuzione.

- È l'**unità di esecuzione** all'interno del S.O.
- Solitamente, l'esecuzione di un processo è **sequenziale** (le istruzioni vengono eseguite in sequenza, secondo l'ordine specificato nel testo del programma)
- Un S.O. Multiprogrammato consente l'esecuzione ***concorrente*** di più processi.

Il 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

Il Concetto di Processo

Processo = {codice, PC, registri, stack, dati}

Inoltre, a un processo possono essere associate delle **risorse** del sistema operativo; ad esempio:

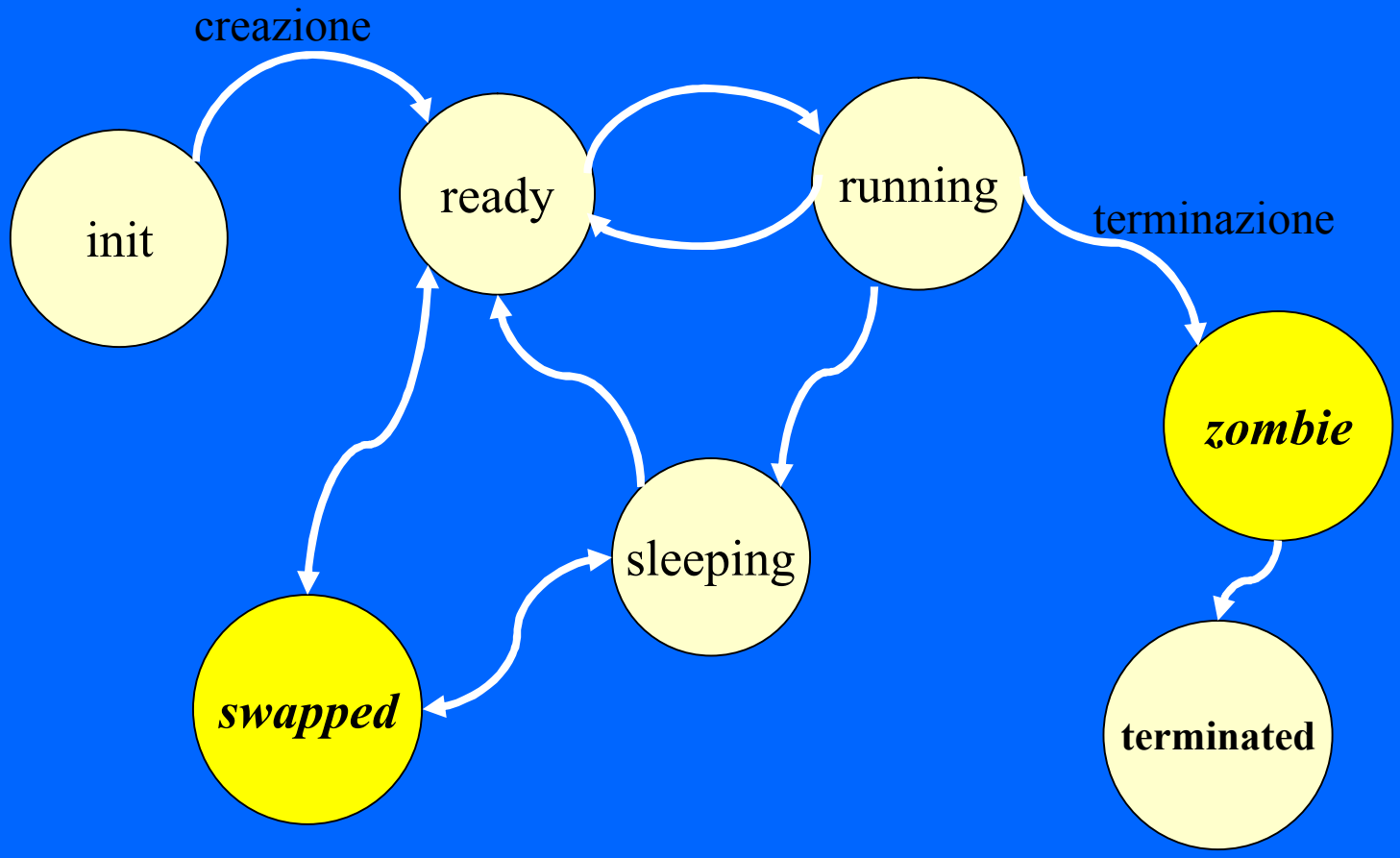
- files aperti
- connessioni di rete
- accesso a dispositivi
- ….

Stati di un processo

Stati di un processo

Transizioni di stato:

Esempio: Unix/Linux

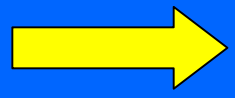


Zombie: il processo è terminato, ma è in attesa che il padre ne rilevi lo stato di terminazione.

Swapped: il processo (o parte di esso) è temporaneamente trasferito in memoria secondaria.

Stati di un Processo

- In un sistema **multiprogrammato/monoprocessore**:
  - 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 nel sistema operativo?

- Ad ogni processo viene associata una struttura dati (descrittore): **Process Control Block (PCB)**.

- Il **PCB** contiene tutte le informazioni relative al processo:
  - Stato del Processo
  - Program Counter
  - Contenuto dei registri di CPU (SP, accumulatori, etc.)
  - Informazioni di scheduling (priorità, puntatori alle code, etc.)
  - Informazioni per la gestione della memoria (registri base, limite, etc.)
  - Informazioni relative all'I/O (risorse allocate, file aperti etc.)
  - Informazioni di accounting (tempo di Cpu utilizzato, etc.)
  - ...

Process Control Block

Il sistema operativo gestisce i PCB di tutti i processi, che vengono collocati all'interno di strutture dati del kernel (tipicamente, *code)*.

Immagine di un processo

L'immagine di un processo è l'insieme delle aree di memoria e delle strutture dati associate al processo.

Tipicamente contiene:

- **codice, dati, PC, registri, stack**
- **Strutture dati del kernel associate al processo:**
  - **PCB del processo**
  - **strutture dati associate a risorse allocate al processo (esempio: tabella dei file aperti)**
  - **..**

Non tutta l'immagine è accessibile in modo *user*:

- parte di **utente:** ***(dati, codice, stack)***
- parte di ***kernel*** ***(PCB+ strutture dati del kernel) -> accesso mediante system calls***

Quando un processo è nello stato *running*, è necessario che la sua immagine sia caricata in memoria;

negli altri stati, sono necessarie soltanto alcune informazioni (di solito, nel PCB):

➔ se necessario, parte dell'immagine di un processo non in esecuzione può essere trasferita in memoria secondaria (*swapping*)

Scheduling dei processi

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

- il caricamento in memoria centrale
- l'assegnazione della CPU

In generale, il sistema operativo compie **tre diverse attività di scheduling:**

- scheduling **a breve termine** (scheduling di CPU)
- scheduling **a medio termine** (swapping)
- [scheduling **a lungo termine** (scheduling dei job)]

Scheduler a lungo termine

Lo scheduler a lungo termine è quella componente del sistema 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** è l'utente che stabilisce direttamente il grado di multiprogrammazione : lo scheduler a lungo termine non è presente.

Scheduler a medio termine (*swapper*)

Il grado di multiprogrammazione non deve essere vincolato dalle esigenze di spazio dei processi

Swapping: trasferimento temporaneo in memoria secondaria di *processi* (di solito sospesi), in modo da consentire l'esecuzione di altri *processi*.

Scheduler a breve termine (o di CPU)

È quella parte del S.O. che si occupa della selezione dei processi pronti a cui assegnare la CPU.

- Nei sistemi time sharing, allo scadere di ogni quanto di tempo, il sistema operativo:
  - 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 al successivo.

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

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

Scheduling e Cambio di Contesto

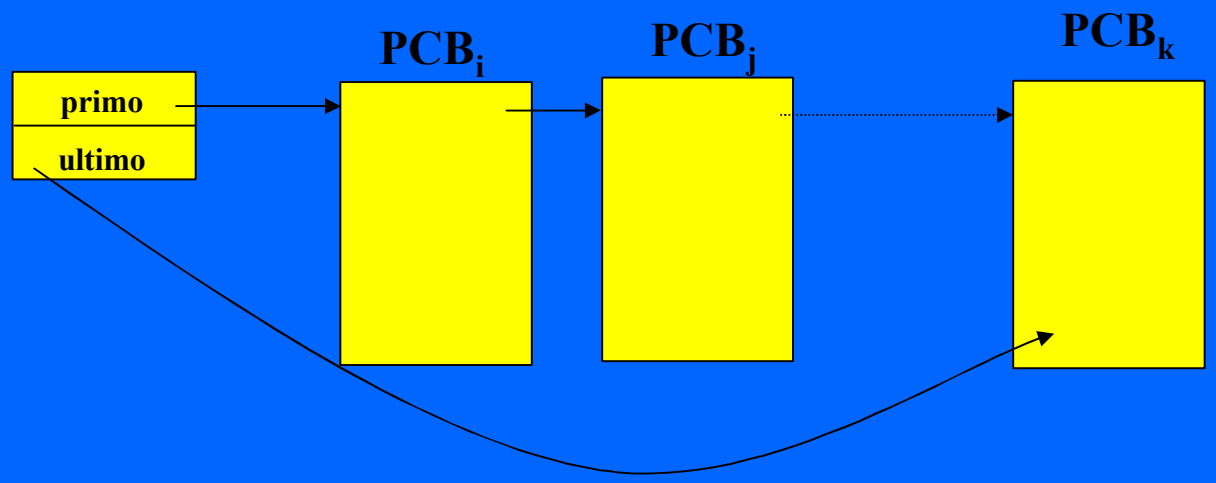
Il cambio di contesto determina un **costo computazionale aggiuntivo** che dipende essenzialmente da:

- **frequenza** di commutazione della CPU ($1/\Delta t$)
- dimensione del **PCB.**
- Costo dei trasferimenti da/verso la memoria
  - esistono sistemi operativi che prevedono **processi *leggeri*** (***threads***), che hanno la proprietà di condividere codice e dati con altri processi:
    - dimensione del PCB ridotta
    - riduzione dell'overhead.

Scheduler della CPU

Lo scheduler a breve termine gestisce :

- la **coda dei processi pronti (*ready queue*):** contiene i PCB dei processi che si trovano nello stato *Ready*.

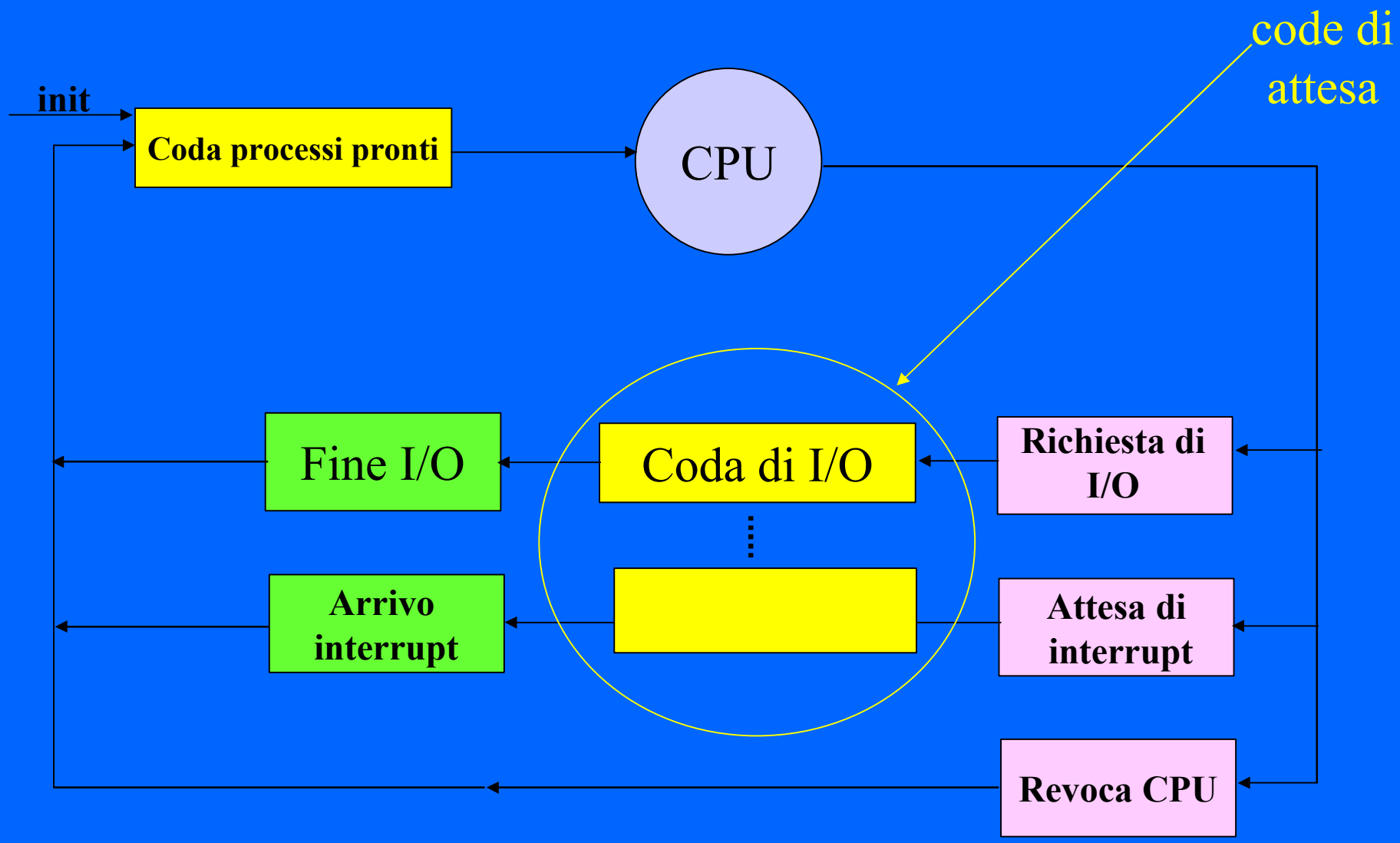


la strategia di gestione della *ready queue* dipende dalle ***politiche*** di scheduling adottate dal S.O.

Code di attesa

Il sistema gestisce I processi sospesi mediante code di attesa (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.

Diagramma di accodamento



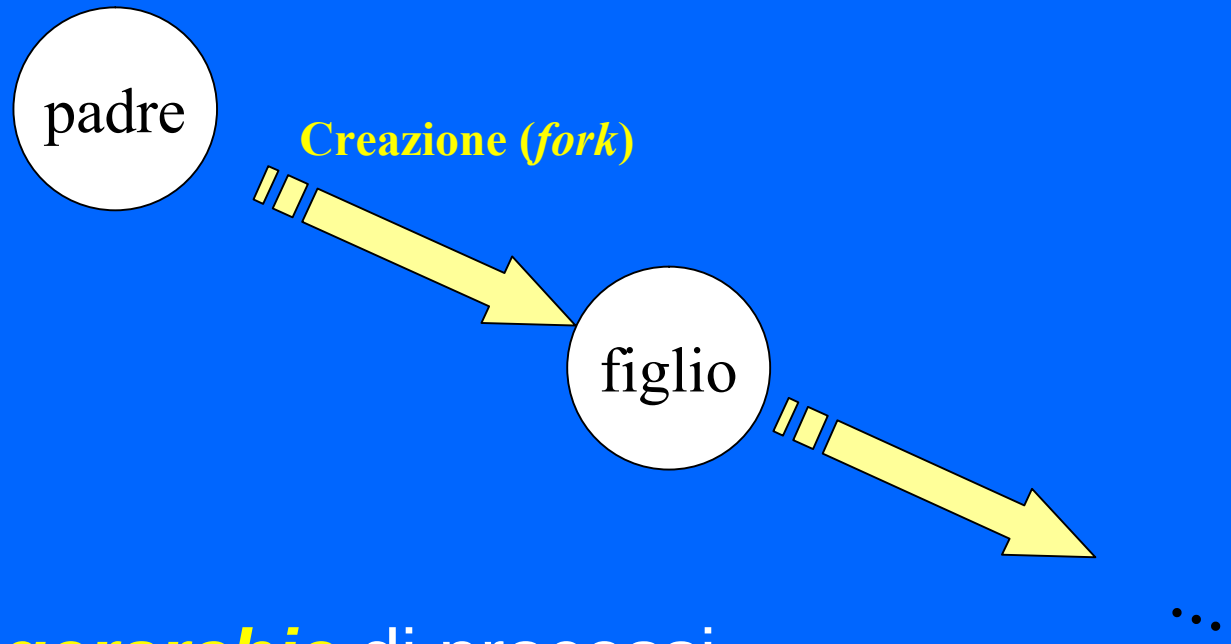
Operazioni sui Processi

Ogni S.O. multiprogrammato prevede dei meccanismi per la gestione dei processi.

- **Meccanismi necessari**:
  - **creazione**
  - **terminazione**
  - **interazione** tra processi
- Sono operazioni **privilegiate** realizzate da *system call*

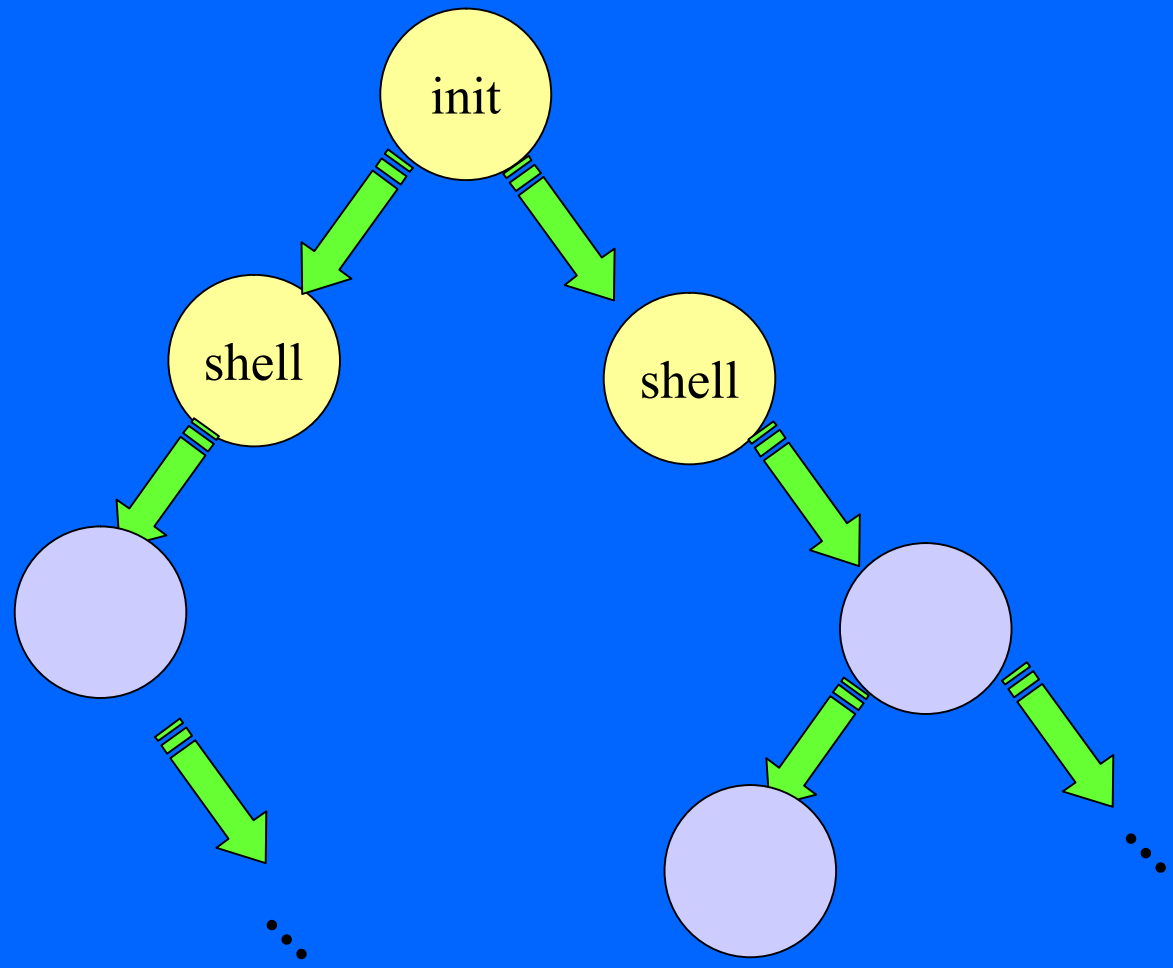
Creazione di Processi

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



➔***gerarchie*** di processi.

Gerarchie di processi (es. Unix)



Relazione Padre-Figlio

Parametri:

- ***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: gli spazi degli indirizzi di padre e figlio contengono codice e dati diversi (es: VMS, exec() in Unix)

Processi Unix

Unix è un sistema operativo *multiprogrammato* a *divisione di tempo*: l'unità di computazione è

il processo.

Caratteristiche del processo Unix:

- **processo pesante con codice *rientrante*:**
  - » **dati non condivisi**
  - » **codice condivisibile con altri processi**
- **funzionamento dual mode:**
  - » **processi di utente (modo *user*)**
  - » **processi di sistema (modo *kernel*)**

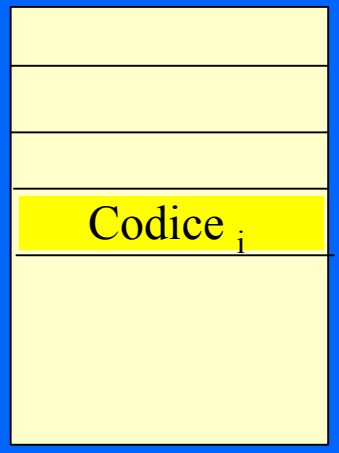
☞ **diverse potenzialità e, in particolare, diversa visibilità della memoria.**

Rappresentazione dei processi Unix

Il codice dei processi è rientrante: più processi possono condividere lo stesso codice (*text*):

- **codice e dati sono separati (modello a *codice puro*)**
- **il S.O. gestisce una struttura dati globale in cui sono contenuti i puntatori ai codici utilizzati ed eventualmente condivisi) dai processi: *text table*.**
- **L'elemento della text table si chiama *text structure* e contiene, ad esempio:**
  - ***puntatore* al codice** (se il processo è *swapped*, riferimento a memoria secondaria)
  - **numero dei processi che lo condividono**

  ...



Text table: 1 elemento ∀ segmento di codice utilizzato

Rappresentazione dei processi Unix

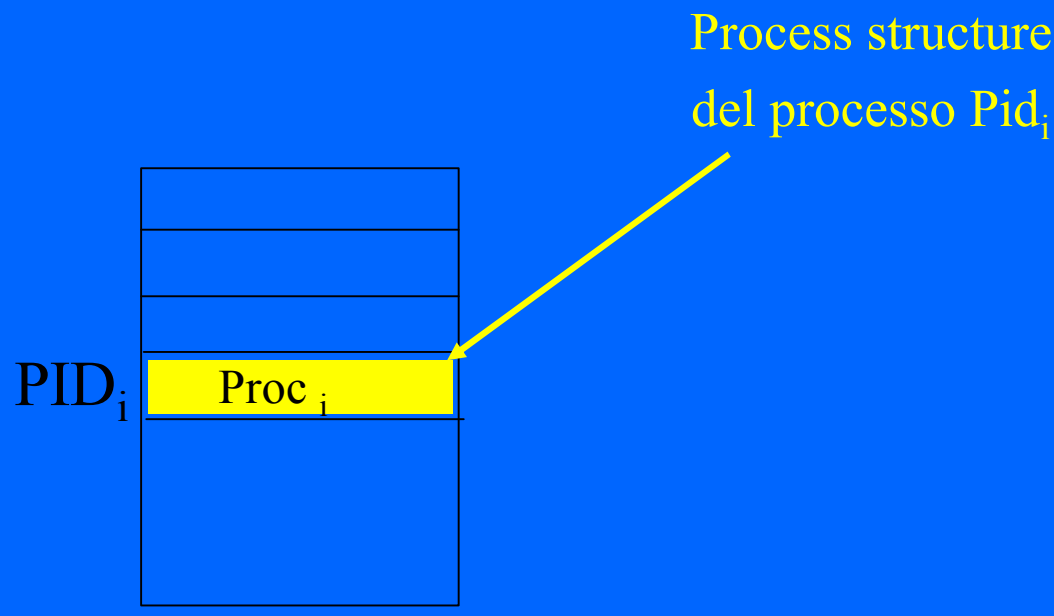
- **Process Control Block**: il descrittore del processo in Unix e` mappato su 2 strutture dati:
  - ***Process Structure:*** **informazioni necessarie al sistema per la gestione del processo (a prescindere dallo stato del processo)**
  - ***User Structure:*** **informazioni necessarie solo se il processo e` residente in memoria centrale**

Process Structure

- contiene, tra l'altro, le seguenti informazioni:
  - » process identifier (**PID**): è un intero positivo che individua univocamente il processo nel sistema
  - » **stato** del processo
  - » puntatori alle varie **aree dati e stack** associati al processo
  - » riferimento indiretto al **codice**: la process structure contiene il riferimento all'elemento della text table associato al codice del processo
  - » informazioni di **scheduling** (es: priorità, tempo di CPU,etc)
  - » Riferimento al **processo padre** (PID del padre)
  - » Informazioni relative alla **gestione di segnali** (segnali inviati ma non ancora gestiti, maschere)
  - » Puntatori a processi successivi in **code** (ad esempio, ready queue)
  - » **Puntatore alla User Structure**

Rappresentazione dei processi Unix

- **Process Structure:** sono organizzate in un vettore: ***Process Table***



Process table: 1 elemento per ogni processo

User Structure

Contiene le informazioni necessarie al S.O. per la gestione del processo, **quando è residente**:

- copia dei **registri** di CPU
- informazioni sulle **risorse allocate** (ad es. **file aperti**)
- informazioni sulla gestione di **segnali** (puntatori a *handler*, etc.)
- **ambiente** del processo: direttorio corrente, utente, gruppo, argc/argv, path, etc.

Immagine di un processo Unix

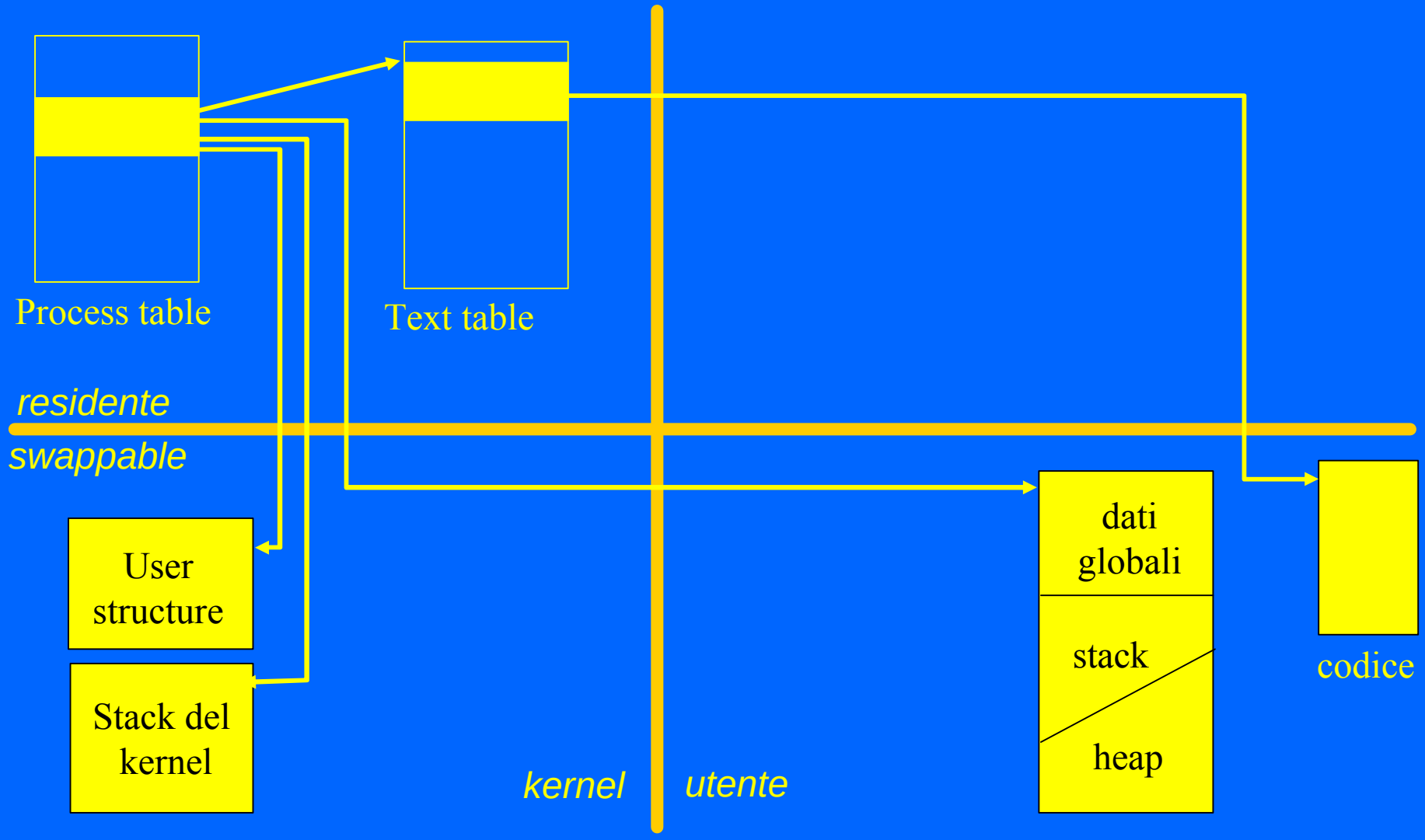


Immagine di un processo Unix

Componenti:

- **process structure**: è l'elemento della process table associato al processo (kernel, residente)
- **text structure**: elemento della text table associato al codice del processo (kernel, residente)
- area **dati globali di utente**: contiene le variabili globali del programma eseguito dal processo (user, swappable)
- **stack, heap** di utente: aree dinamiche associate al programma eseguito (user, swappable)
- **stack del kernel**: stack di sistema associato al processo per le chiamate a system call (kernel, swappable)
- **user structure**: struttura dati contenente i dati necessari al kernel per la gestione del processo quando è residente (kernel, swappable).

Rappresentazione di processi Unix

- **Process Structure**: informazioni necessarie per la gestione del processo, anche se questo è ***swappato*** in memoria secondaria.
- **User Structure**: il suo contenuto è necessario **solo in caso di esecuzione** del processo (stato *running*

PCB = Process Structure + User Structure

Processi "*pesanti*"

Processo = {codice, PC, registri, dati, stack..}

Quindi:

- ***processi*** diversi eseguono codici distinti
- ***processi*** diversi accedono a dati diversi

➔ i ***processi*** **non condividono memoria**

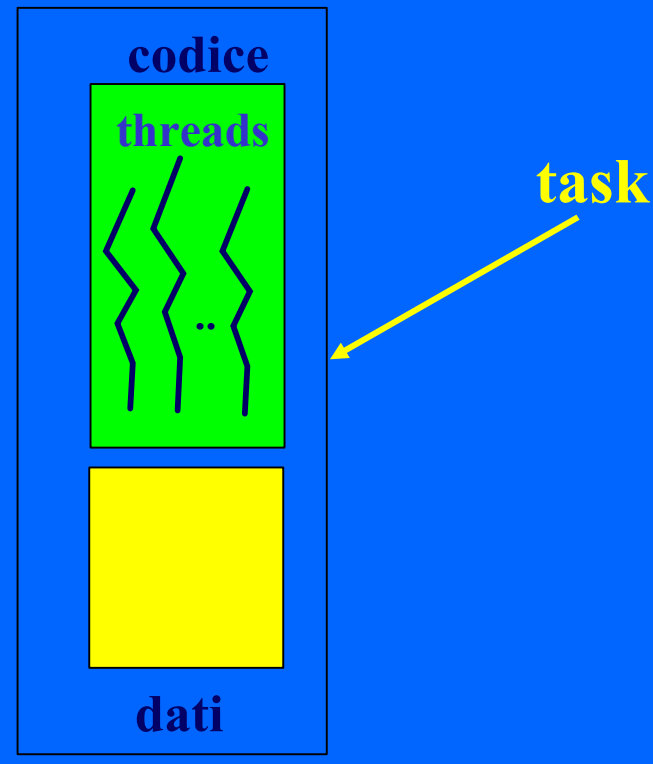
Il costo del cambio di contesto per processi pesanti può essere elevato (dimensione del descrittore, accessi in memoria).

"PROCESSI PESANTI" (heavy-weight processes)

Processi “leggeri” (threads)

Un *thread* è un’unità di esecuzione che **condivide lo spazio di indirizzamento (dati e codice)** con altri *thread* ad esso associati.

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



Thread

Un thread rappresenta un flusso di esecuzione all'interno di un processo pesante.

- Multithreading: molteplicità di flussi di esecuzione all'interno di un processo pesante.
- Tutti i thread definiti in un processo **condividono** le risorse del processo, risiedono nello stesso spazio di indirizzamento ed hanno accesso agli stessi dati.

Ogni thread ha :

- uno **stato** di esecuzione (running, ready, blocked)
- un **contesto** che è salvato quando il thread non è in esecuzione
- uno stack di esecuzione
- spazio di memoria statico per le variabili locali
- accesso alla memoria e alle risorse del processo pesante condiviso con gli altri thread.

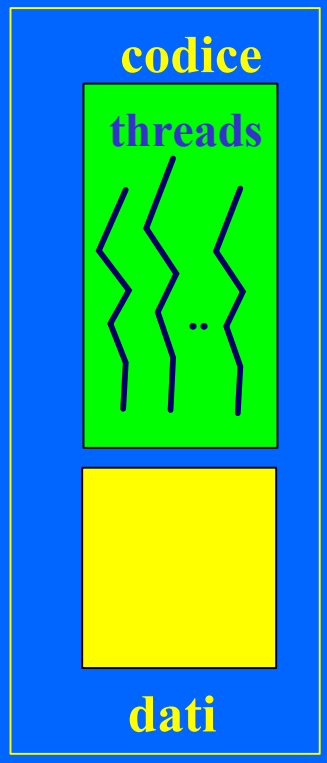
Thread

- codice e dati non sono caratteristiche del singolo thread, ma del task al quale appartiene:

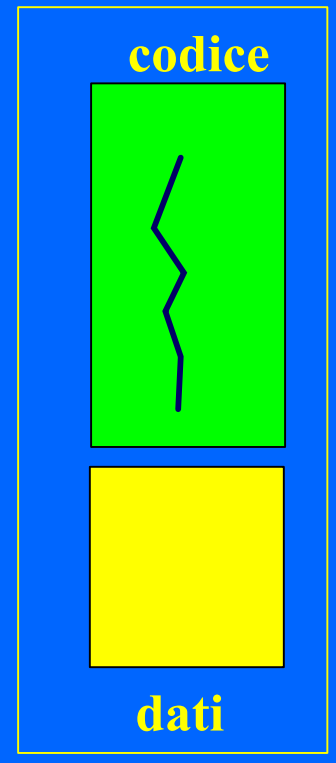
  Thread= {PC, registri, stack}

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

Processi leggeri (threads)



Applicazione multi-threaded.



Il **processo pesante** equivale a un task con un solo thread.

Proprieta` dei Threads

- **Condivisione di memoria:** a differenza dei processi (*pesanti*) , un thread può condividere variabili con altri (appartenenti allo stesso task).
- **Minor costo di context switch:** il PCB di un thread non contiene alcuna informazione relativa a codice e dati
  - ➔ il cambio di contesto **tra thread dello stesso task** ha un costo notevolmente inferiore al caso dei processi pesanti.
- **Minor protezione:** threads appartenenti allo stesso task possono modificare dati ***gestiti*** da altri threads.

Vantaggi

- maggiore efficienza: le operazioni di context switch, ed in generale di estione dei thread (creazione etc.) sono più economiche rispetto al caso dei processi.
- maggiori possibilità di utilizzo di architetture multiprocessore.

Realizzazione di Threads

Alcuni sistemi operativi realizzano il concetto di thread (ad es: NT, GNU/**linux**, Solaris).

Possibilità di realizzazione:

- **A livello utente (**Andrew - Carnegie Mellon, Java**)**
- **A livello kernel (NT, OS2, LINUX, L4, Hurd)**
- **A entrambi i livelli (Solaris)**

Realizzazione a livello utente

I thread vengono realizzati mediante librerie di funzioni che operano a livello **utente** e che forniscono il supporto alla *gestione e* sincronizzazione dei thread.

- Il sistema operativo *ignora la presenza dei thread* continuando a gestire solo i processi.
- il passaggio da un thread al successivo (nello stesso task) non richiede interruzioni al S.O. (**maggior rapidità**)
- il S.O. ***vede* processi *pesanti***: **minore efficienza**
  - es: sospensione di un thread
  - Cambio di contesto tra thread di task diversi

Realizzazione a livello kernel

Il kernel del sistema si fa carico di tutte le funzioni per la gestione dei thread.

- A ciascuna funzione corrisponde una ***system call***.
- Quando un thread si *blocca,* il S.O. può mettere in esecuzione un altro thread dello *stesso processo*.
- Soluzione *meno efficiente* della precedente.
- Possibilità di eseguire thread diversi appartenenti allo stesso processo su unità di elaborazione differenti (*architettura multiprocessore).*
- il S.O. gestisce direttamente i cambi di contesto:
  - tra threads dello stesso task (trasferimento di registri)
  - tra task
- il S.O. fornisce strumenti per la sincronizzazione nell'accesso di threads a variabili comuni.

Realizzazione di Threads

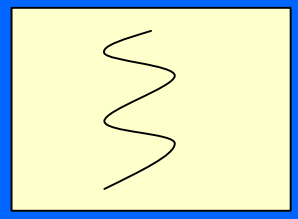
Soluzioni Miste (es: Solaris 2):

- i threads possono essere realizzati a entrambi i livelli.
- Creazione di thread, politiche di assegnazione della CPU e sincronizzazione a *livello utente.*
- I thread a livello utente sono mappati in un numero (minore o uguale) di thread a livello nucleo.

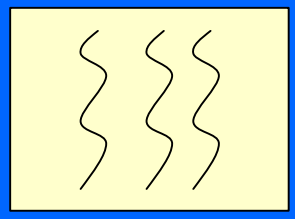
Vantaggi:

- Thread della stessa applicazione possono essere eseguiti in parallelo su processori diversi.
- Una chiamata di sistema bloccante non blocca necessariamente lo stesso processo.

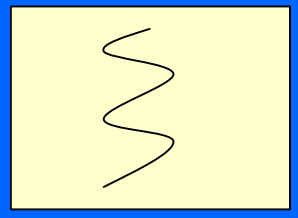
Realizzazione di Threads

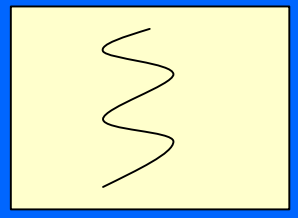


un thread per processo

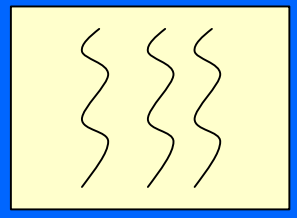


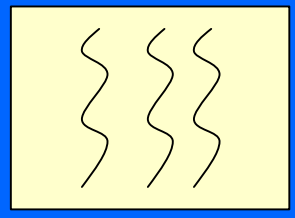
più thread per processo





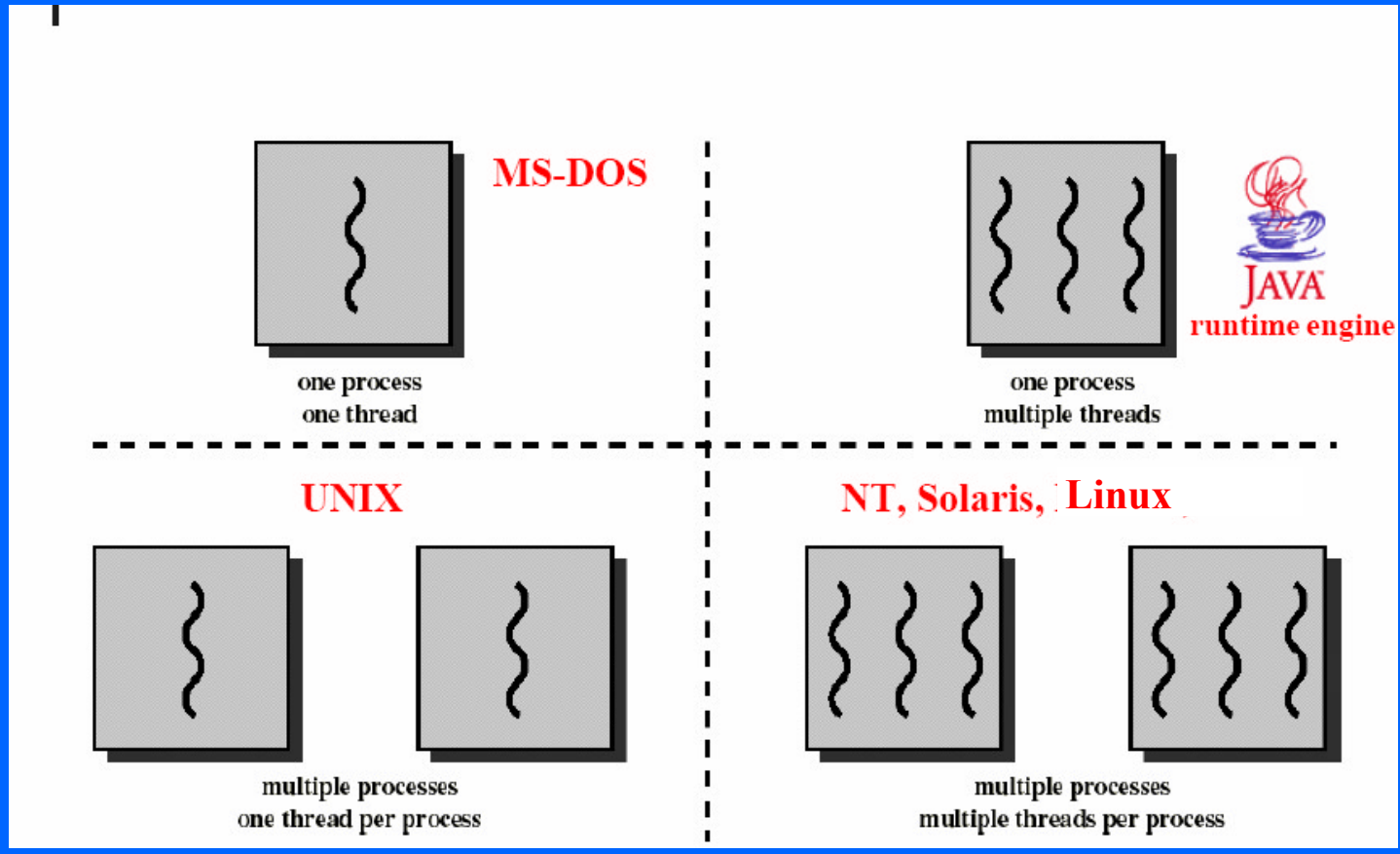
Processi multipli:
un thread per processo





Processi multipli;
più thread per processo

Realizzazione di Threads



Linux: caratteristiche dei thread

In Linux l'unità di esecuzione è il thread.

Caratteristiche:

- Il thread e` realizzato **a *livello kernel*** (e` l'unita` di schedulazione)
- I thread vengono creati all'interno di un processo (task) per eseguire una funzione
- Ogni thread ha un PID che lo identifica univocamente :
  - Il sistema non realizza esplicitamente il task
  - Implicitamente: il task è il gruppo di thread discendenti da uno stesso thread
- Il kernel non "vede" i task, ma gestisce un insieme *piatto* di thread:difficoltà nella realizzazione di operazioni riferite a task (per esempio, invio di segnali a task)
- Gestione di thread:
  - System call native (ad esempio **`clone`, per la creazione di thread): scarsa portabilità**
  - **Uso di librerie standard (Posix conformant): `pthread`**

- Sincronizzazione (Posix):in aggiunta ai meccanismi tipici di Unix, sono disponibili strumenti per sincronizzare i thread negli accessi a dati condivisi: **Lock**, **Semafori,Variabili condizione** .

Interazione tra processi

I processi *possono*, in generale, 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:** l'interazione consiste nello scambio di informazioni, al fine di eseguire un'attività comune.
- **Competizione:** i processi interagiscono per sincronizzarsi nell'accesso a risorse comuni.

Processi interagenti

Supporto all'interazione:

l'interazione può avvenire mediante:

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

Riferimenti Bibliografici

- Processi e thread:
  - Silbershatz, Galvin: Capitolo 4
  - Stallings: Capitoli 3 e 4
- Processi Unix:
  - Stevens: Advanced Programming in the Unix Environment