

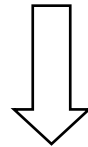
I PROCESSI

Che cos'è un processo...

Un processo è un programma in esecuzione

PROGRAMMA : entità statica

PROCESSO : entità dinamica

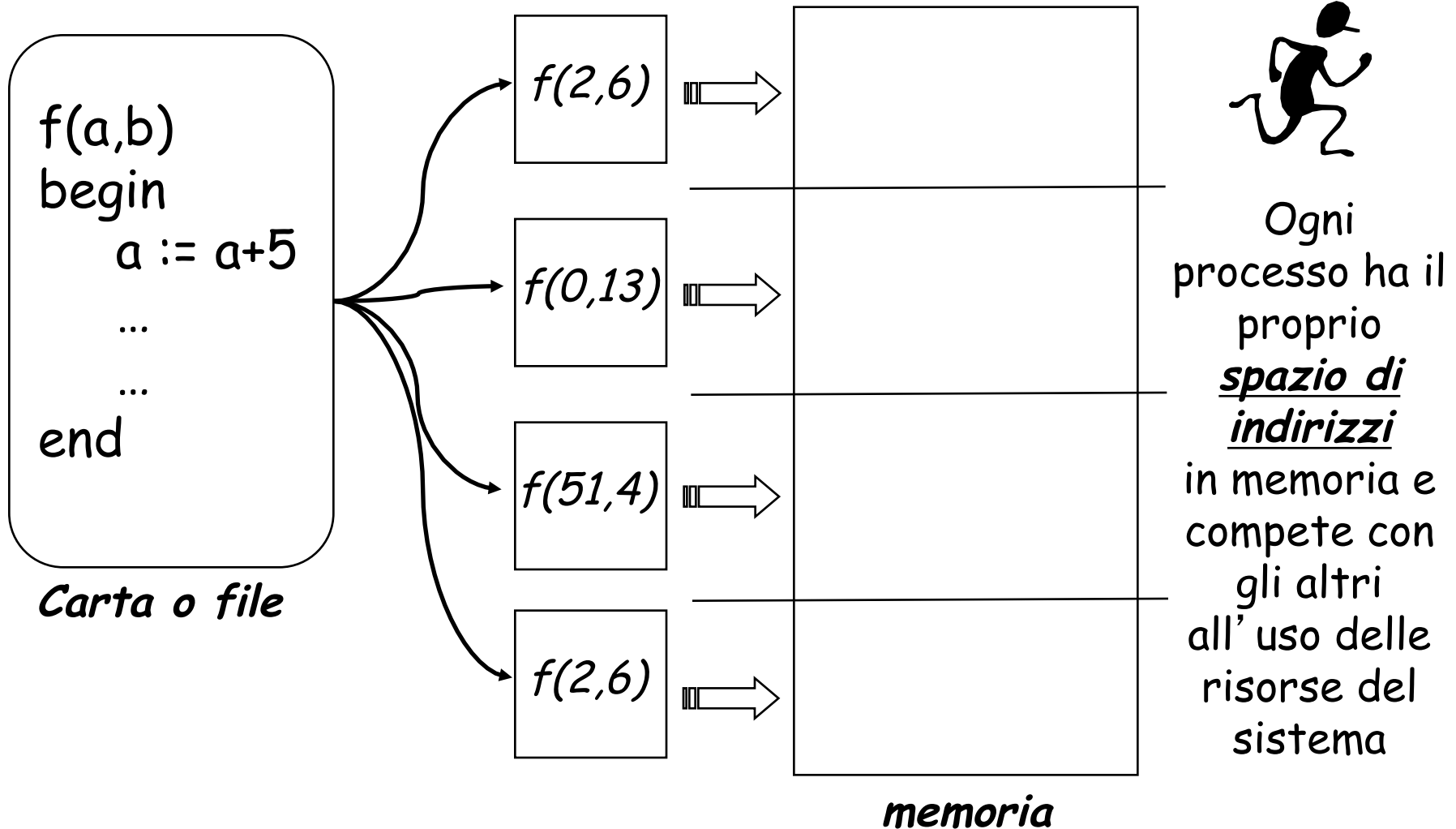


Ad un solo programma può corrispondere più di un processo

Ad un solo processo può corrispondere un solo programma

... esempio di molteplicità ...

PROGRAMMA → PROCESSI



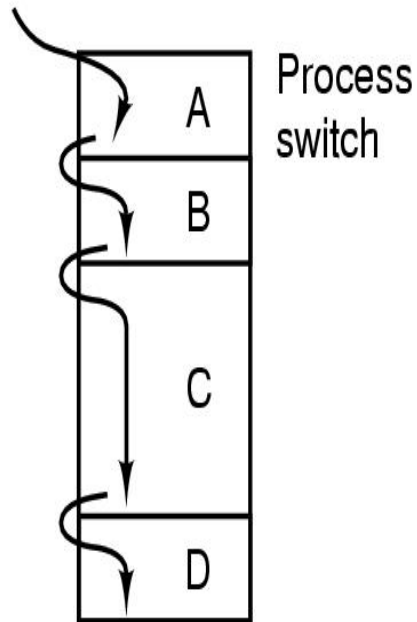
... e i suoi elementi caratterizzanti

- sezione testo (= codice)
- program counter
- (valori dei) registri della CPU
- stack
- sezione dati
- (*stato*)

Un processo esegue *un'istruzione alla volta*, in maniera *sequenziale*

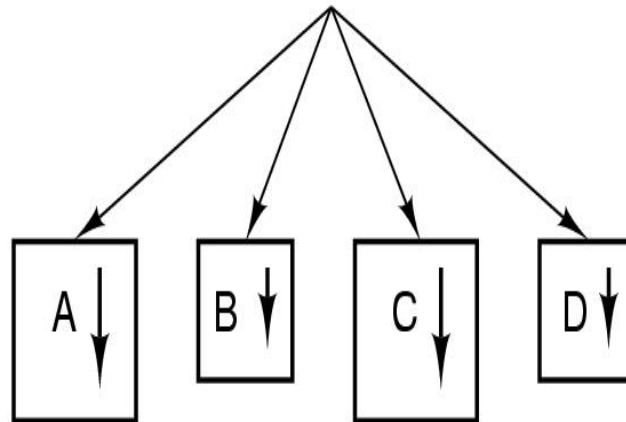
Sempre un'unica CPU!

One program counter

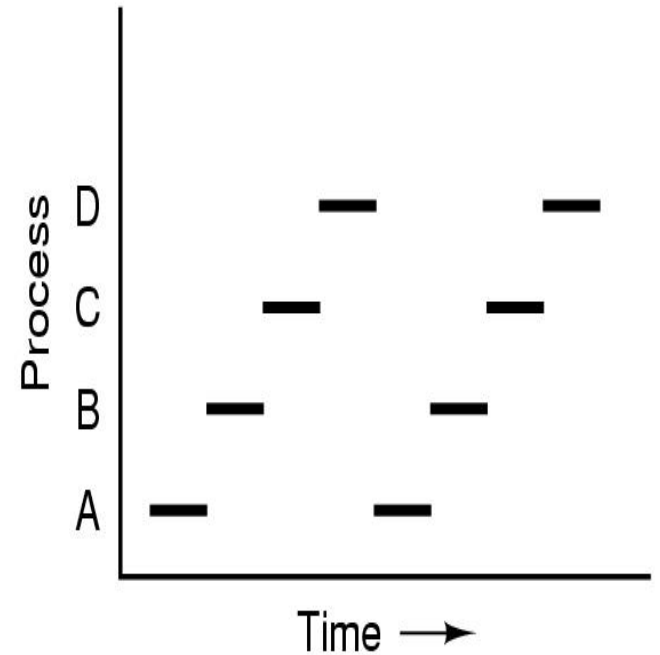


(a)

Four program counters

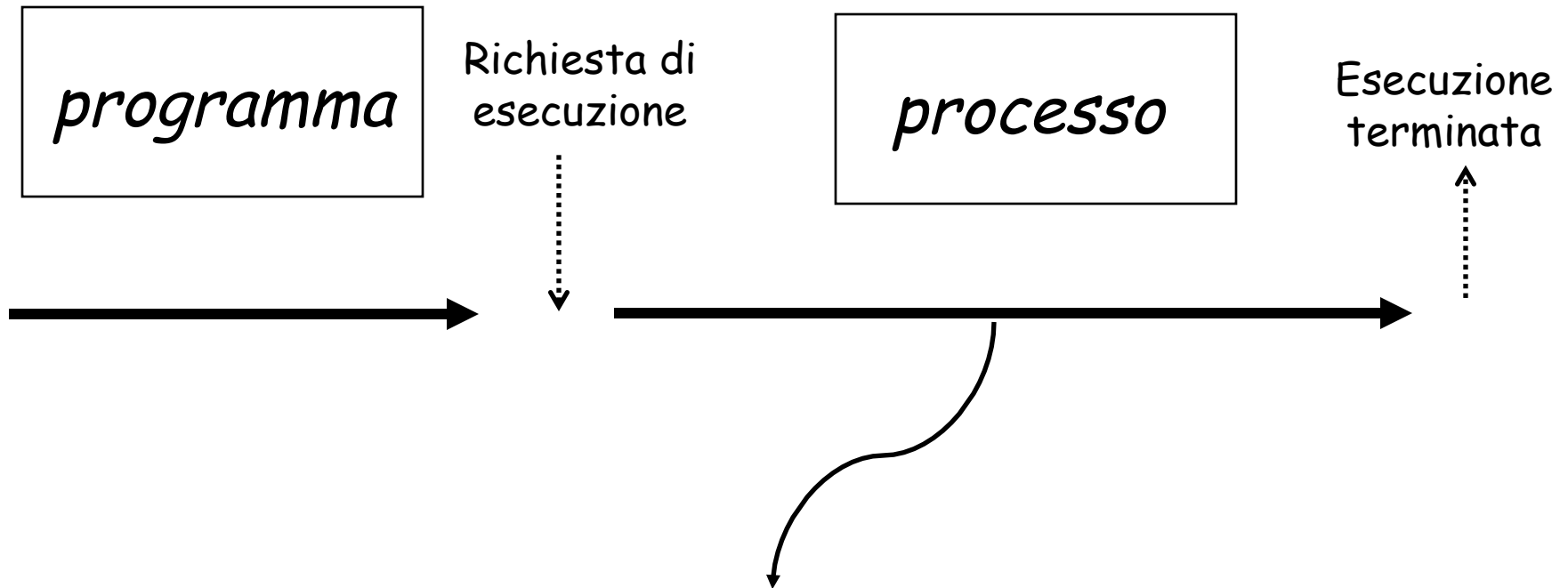


(b)



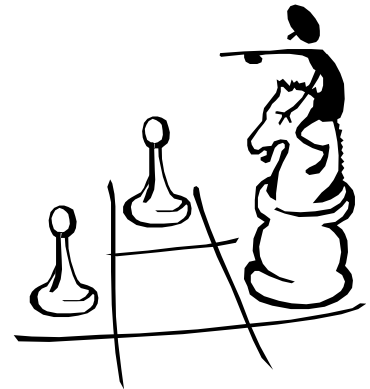
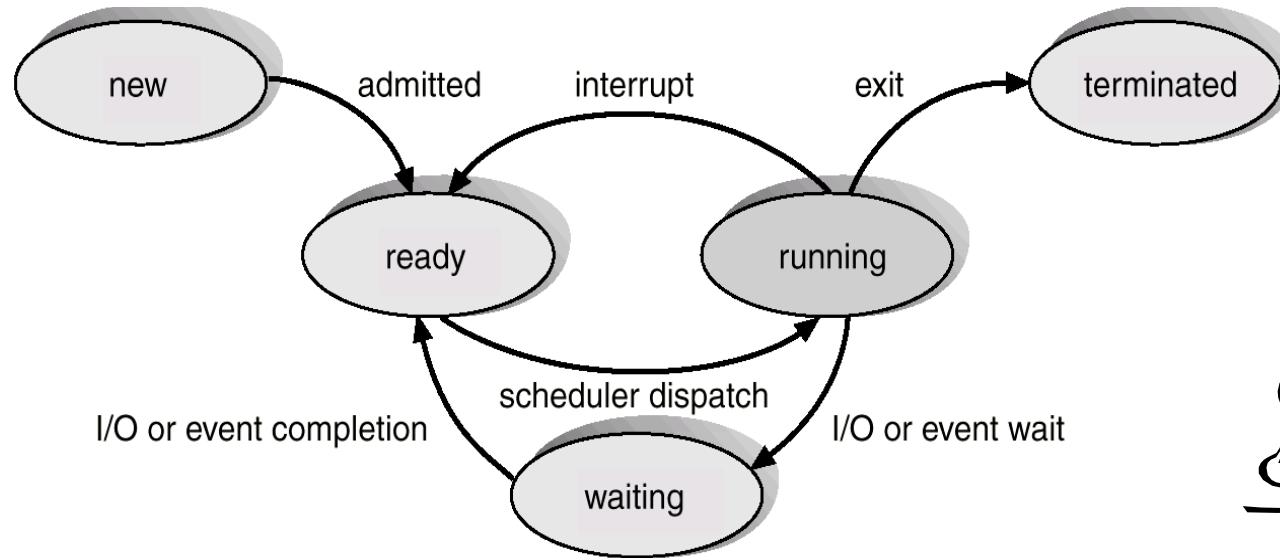
(c)

“Vita” di un processo



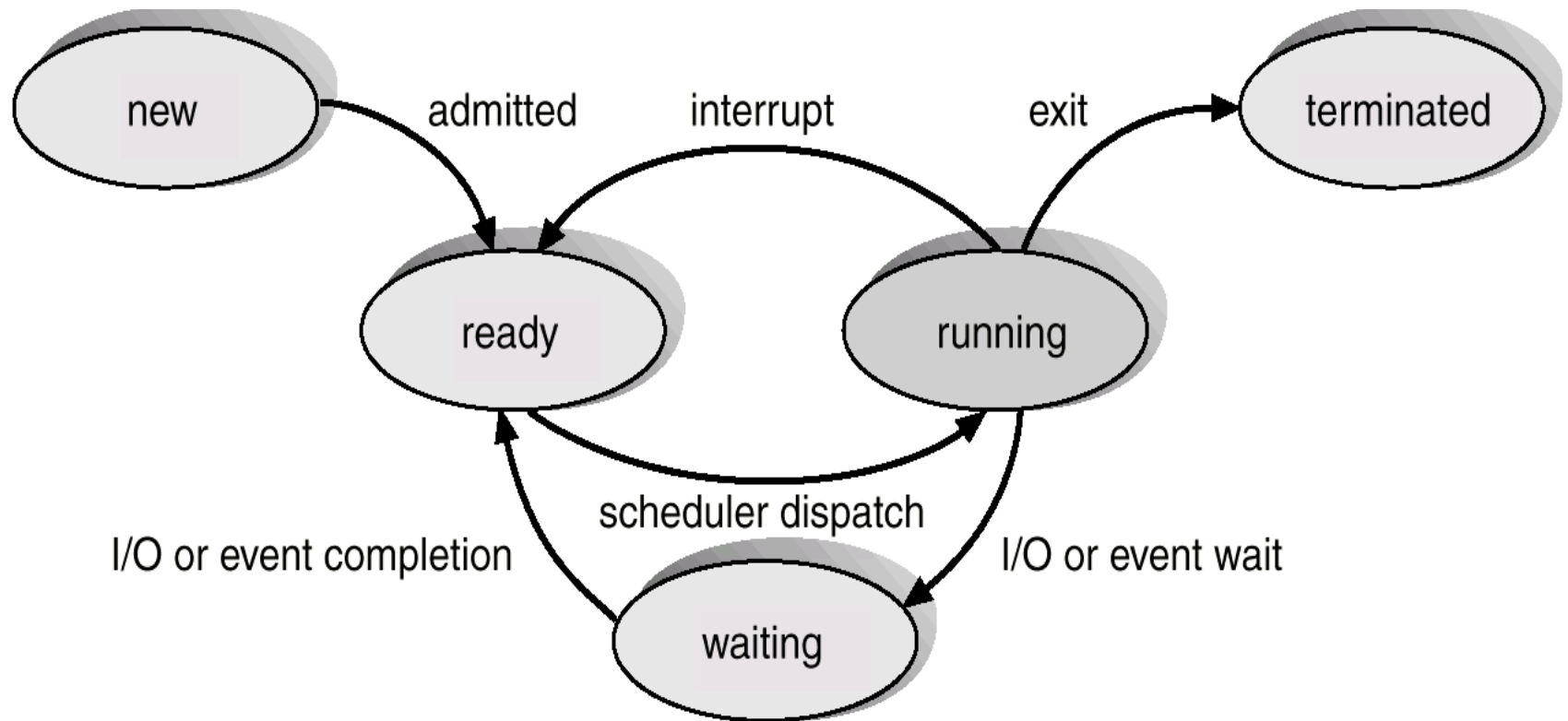
Durante la sua “vita” un processo cambia stato

Cambiamenti di stato



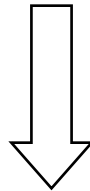
<i>new</i>	il processo e' stato appena creato
<i>ready</i>	il processo e' pronto ad essere eseguito
<i>running</i>	il processo e' in esecuzione
<i>waiting</i>	il processo sta attendendo che accada "qualcosa"
<i>terminated</i>	il processo ha finito la sua esecuzione

***Quanti, dei processi presenti nel sistema,
possono stare in ognuno degli stati
possibili (0,1,n) ?!***



Multitasking (time-sharing) ...

In un sistema di calcolo e' *in vita*
piu' di un processo alla volta e
questi condividono risorse



OS deve preoccuparsi di gestire
le risorse (ex. CPU) in maniera
“opportuna”

... e concetti “vicini”

Multiprogrammazione - piu' di un processo alla volta nel sistema, ma *non necessariamente c'è competizione per le risorse* :

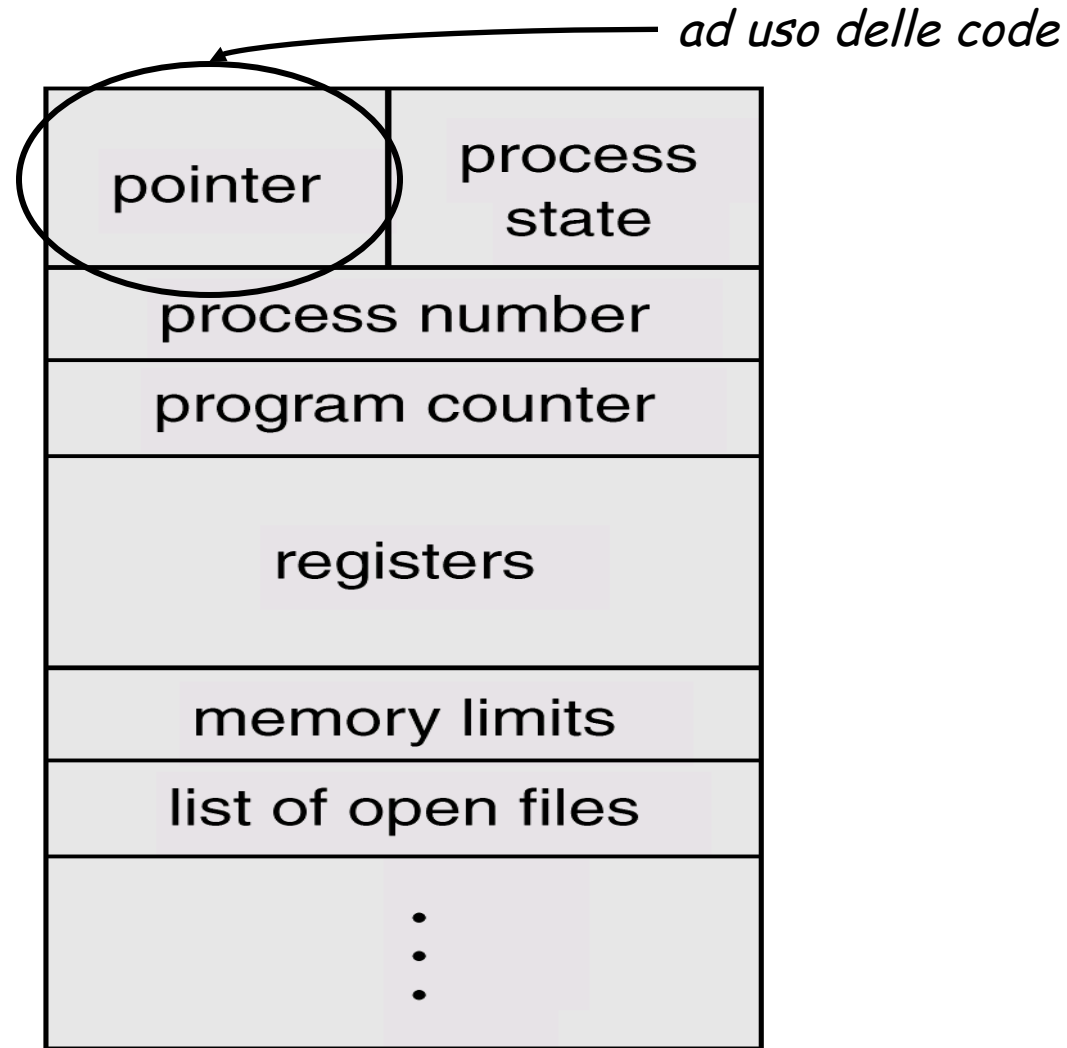
puo' essere eseguito prima interamente un processo e poi viene dato il controllo al successivo

Multiutenza : piu' *utenti* possono utilizzare contemporaneamente il sistema

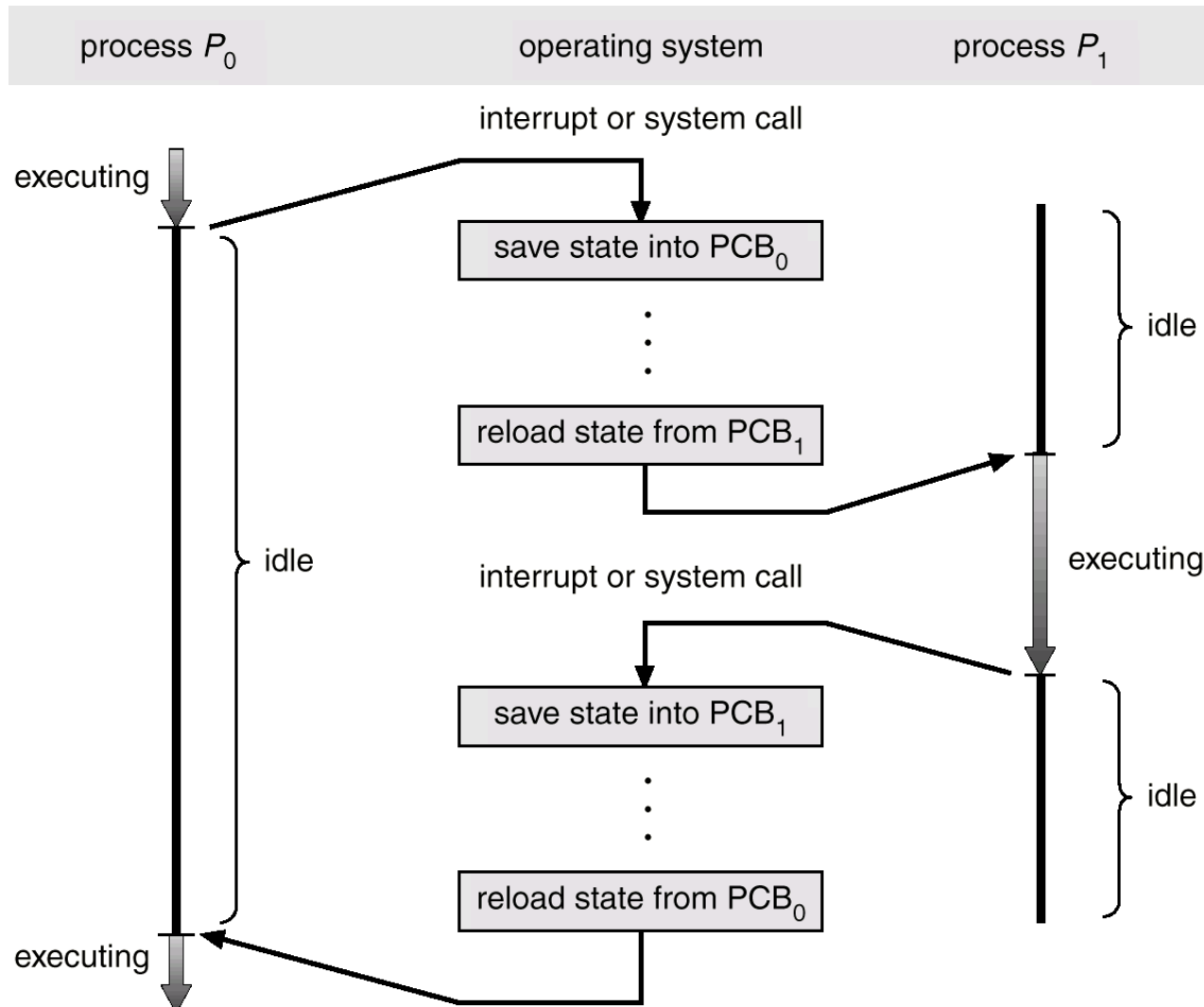
?!

Rappresentazione di un processo: *Process Control Block (PCB)*

- Identificatore (= numero)
- Stato del processo
- Program counter
- Registri della CPU
- Informazioni di scheduling
- Informazioni di memoria
- Informazioni di I/O
- Contabilita' uso risorse



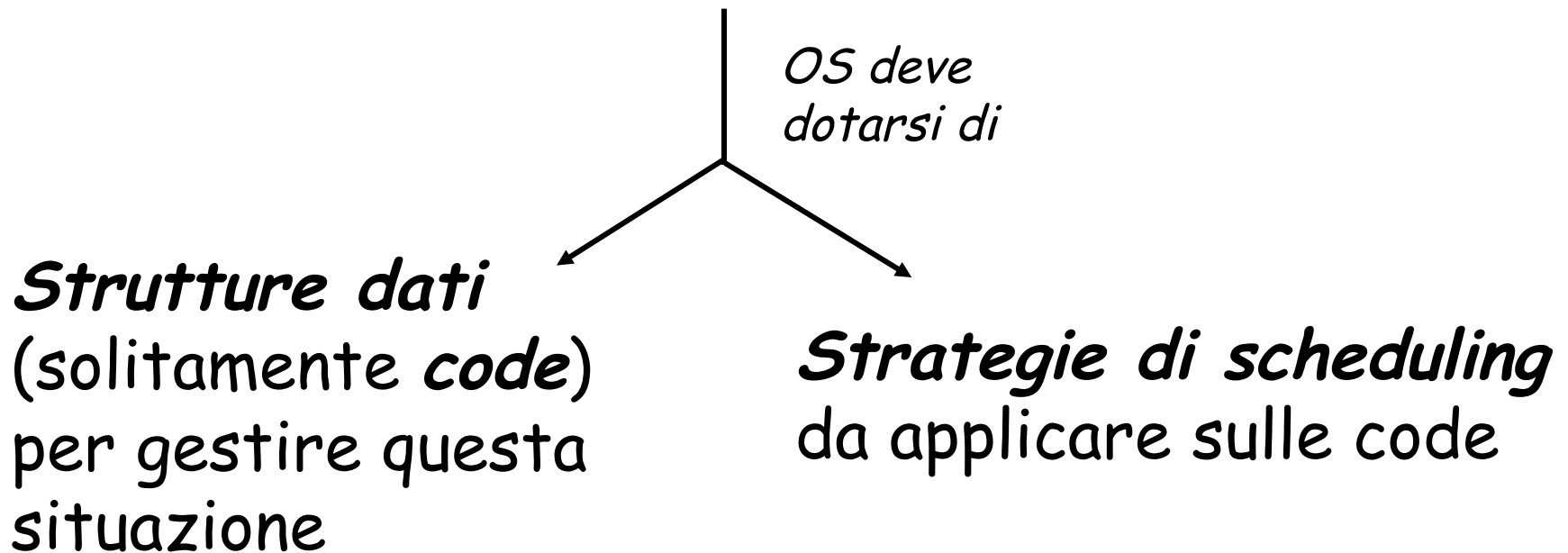
Condivisione della CPU tra processi



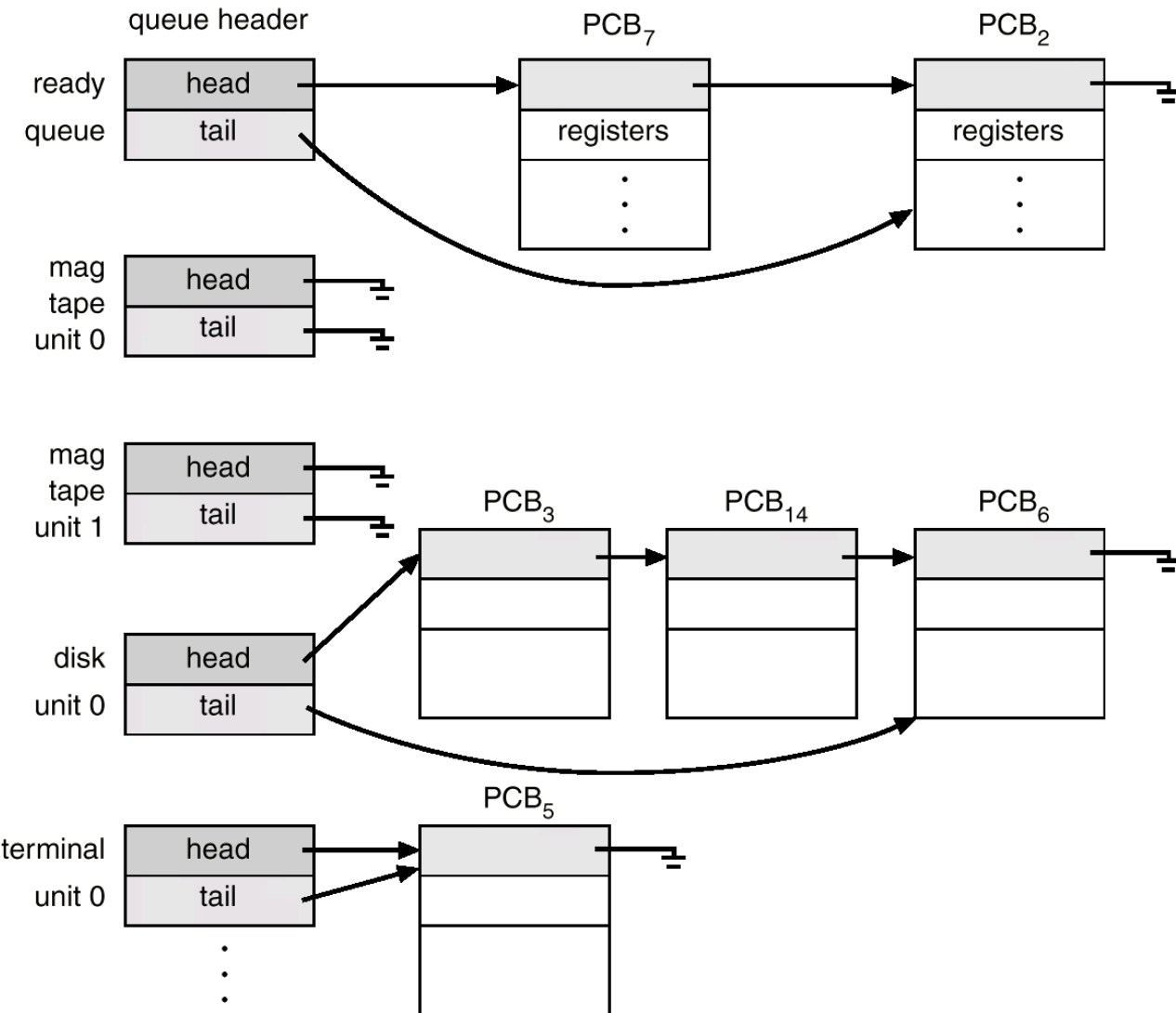
Code di processi nel sistema

OS deve gestire piu' processi alla volta

Piu' processi possono voler usare la stessa risorsa contemporaneamente

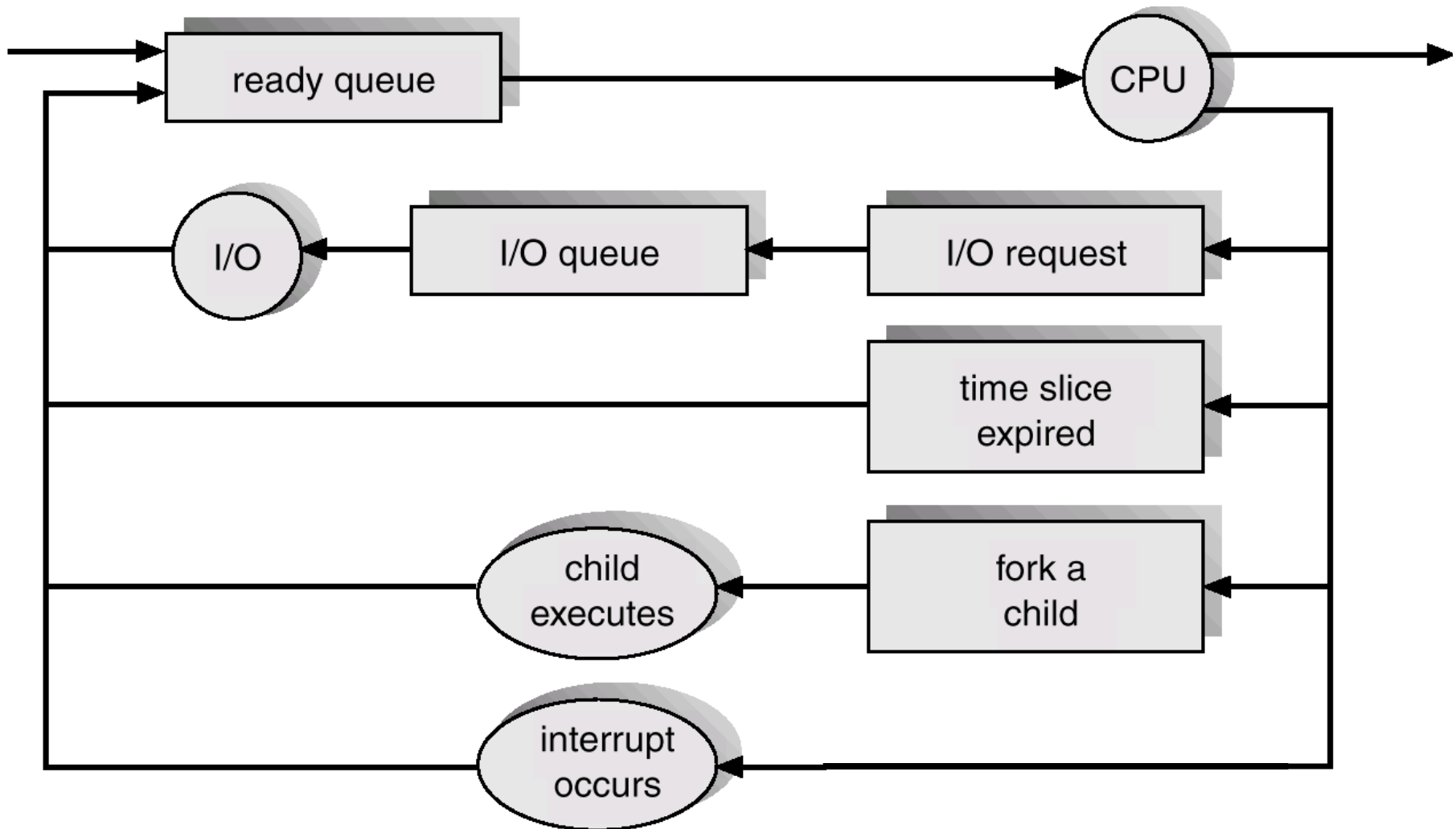


Tipi di code...

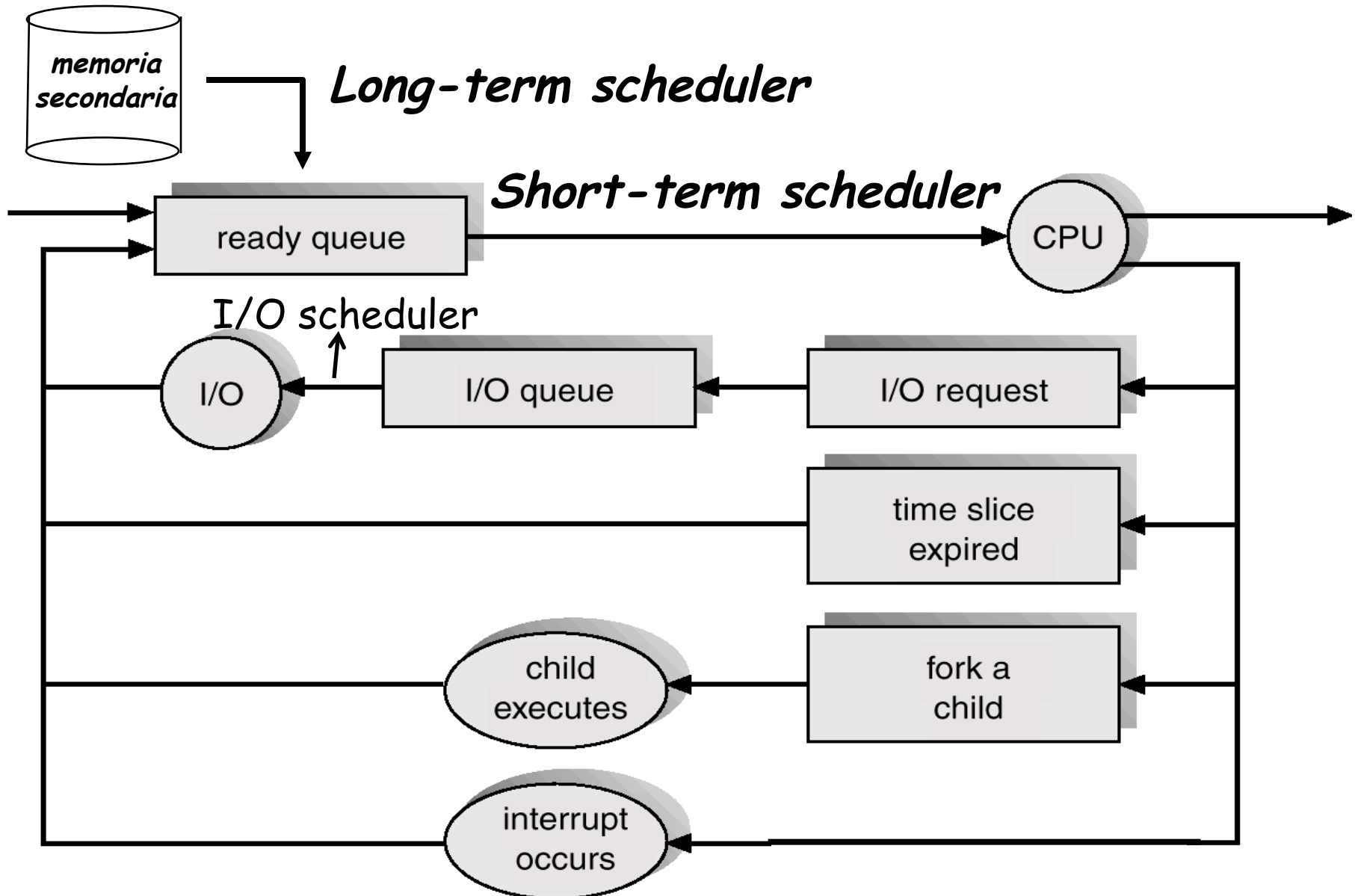


- **Ready queue** - insieme dei processi che risiedono *in memoria principale*, pronti in attesa di esecuzione
- **Coda di dispositivo** - insieme di processi che *attendono l'utilizzo* di un certo dispositivo (*dove sono memorizzati?!*)

... e possibili migrazioni tra code (dettate da cambiamenti di stato)

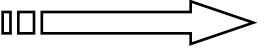


Dove sono i principali *OS schedulers* ...



... e cosa fanno

Short-term scheduler (o CPU scheduler)

- Seleziona il prossimo processo da *mandare in esecuzione*
- Viene invocato molto frequentemente (in millisecondi) \Rightarrow *deve essere veloce!!!* 

Long-term scheduler

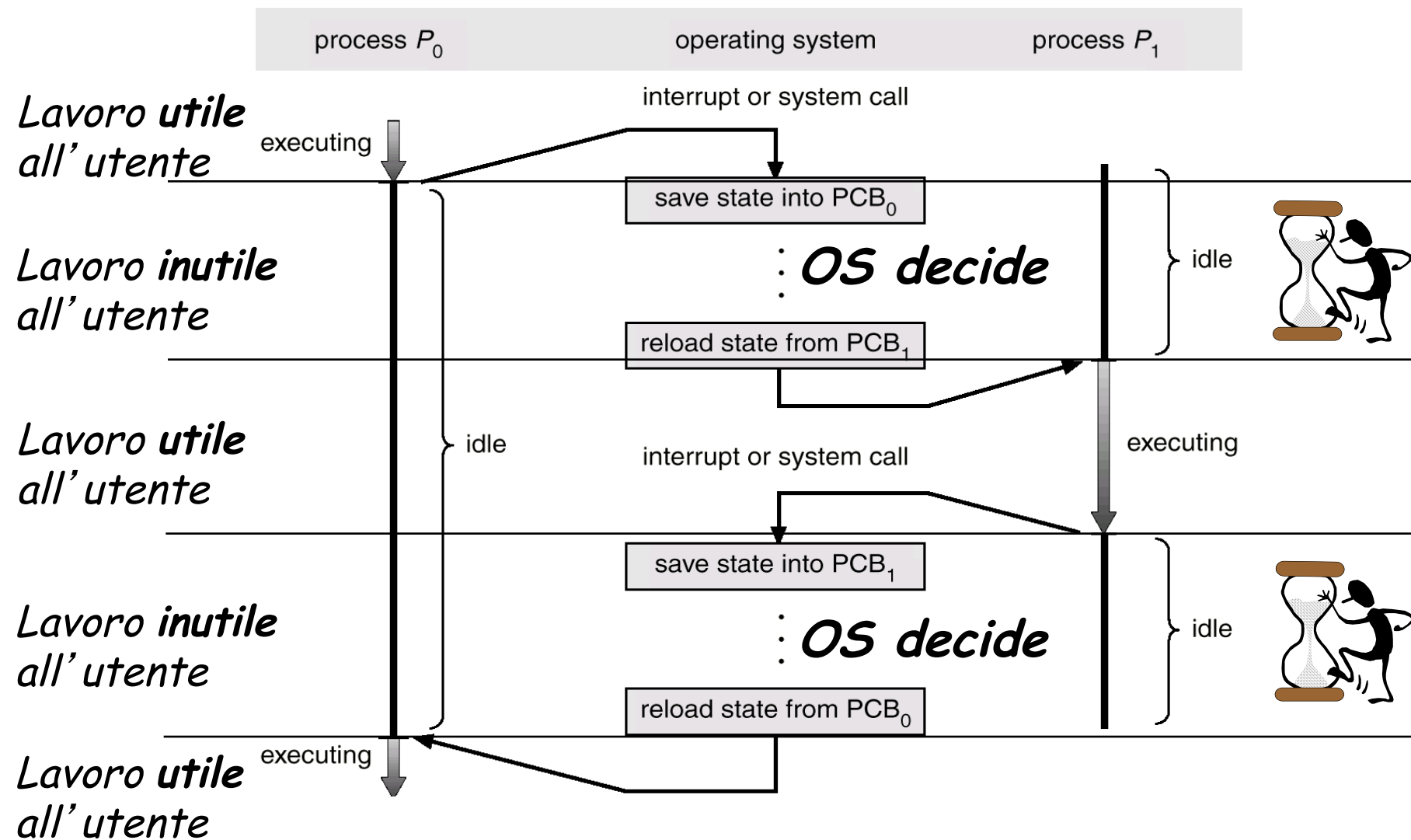
- Seleziona quali processi devono risiedere *nella ready queue* rispetto a tutti quelli pronti nel sistema
- Controlla quindi il *grado di multiprogrammazione* del sistema (giusto bilanciamento tra *CPU-* e *I/O-bounded*)
- Viene invocato molto infrequentemente (in secondi o minuti) \Rightarrow *puo' anche essere lento*

Context Switching

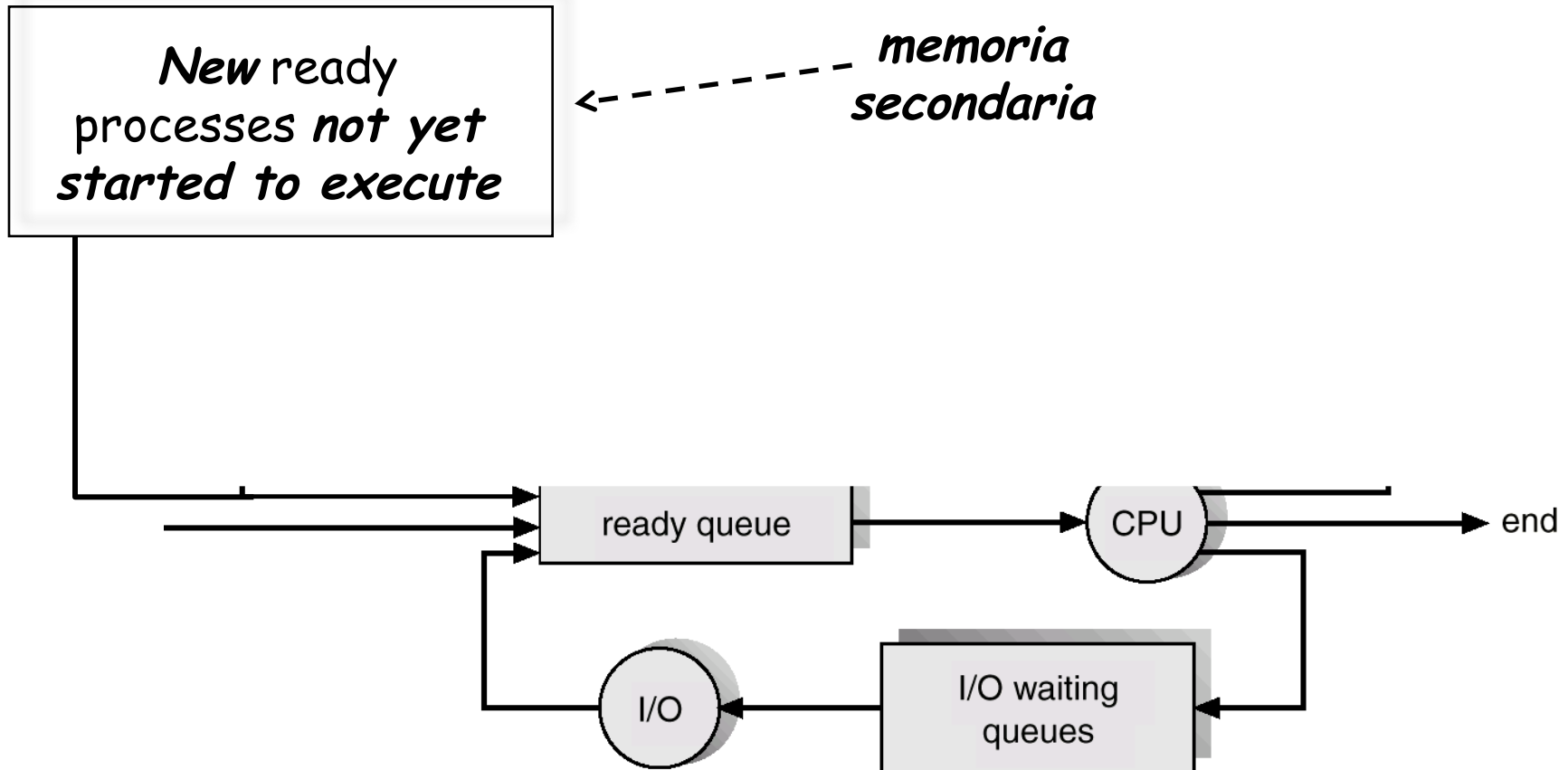
L'operazione di *avvicendamento* di un processo in esecuzione con un altro

- OS deve salvare il PCB del vecchio processo e caricare quello del nuovo
- Queste operazioni portano via una quantita' di tempo che dipende anche dal supporto hardware
- In questo tempo OS non sta facendo *lavoro utile* per l'utente, si tratta quindi di *overhead*

Overhead di short-term scheduling

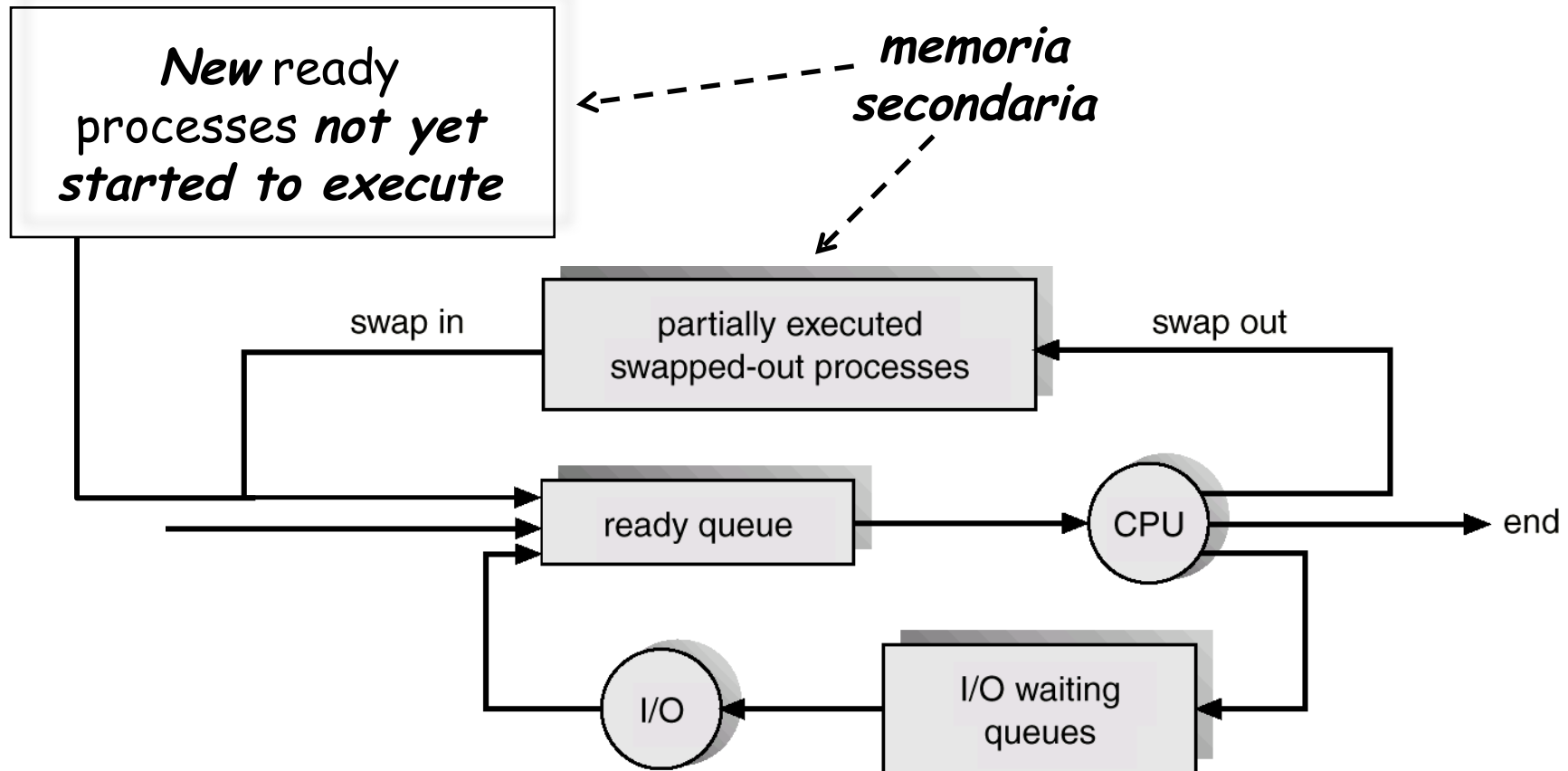


Long-term scheduling



Long-term : la decisione si prende quando un processo finisce e lascia quindi la ready queue

... e Medium-term scheduling

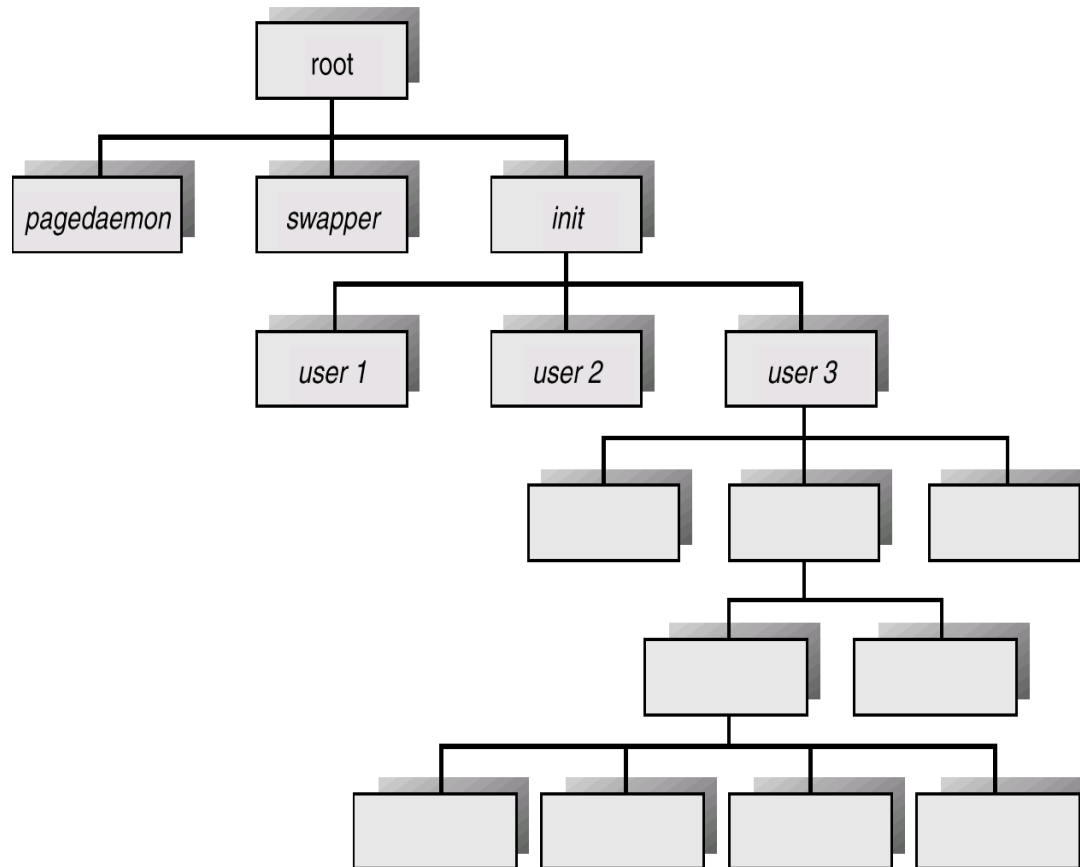


Long-term : la decisione si prende quando un processo finisce e lascia quindi la ready queue

Medium-term : la decisione si prende “ogni tanto”,
quando **necessario!!! (*swapping*)**

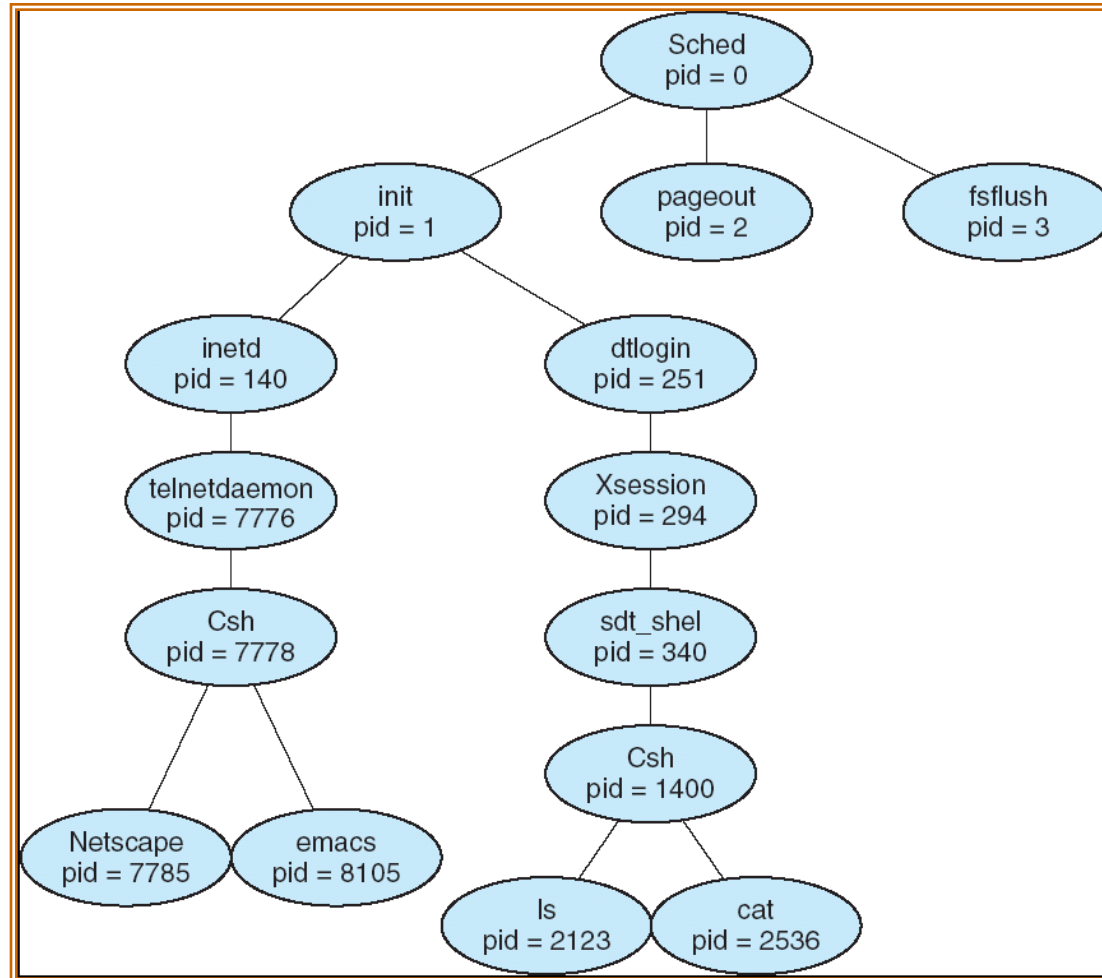
Creazione di un processo

Un processo (*padre*) crea altri processi (*figli*) che, a loro volta, possono creare altri processi, il tutto a formare un *albero di processi*



Esempio:
UNIX

Altro esempio di albero di processi



La system call “*fork*” di UNIX ...

fork crea un nuovo processo che ha una copia dello spazio di indirizzi del padre:

- ripartono dalla stessa istruzione dopo la **fork**, cioè si origina *un nuovo program counter* all'interno dello stesso codice
- per poterli distinguere il *codice di ritorno* della chiamata alla **fork** è *differente* tra il padre e il figlio

... un esempio d'uso ...

```
while (TRUE) {  
    type_prompt( );  
    read_command (command, parameters)  
  
    if (fork() != 0) {  
        /* Parent code */  
        waitpid( -1, &status, 0);  
    } else {  
        /* Child code */  
        execve (command, parameters, 0);  
    }  
}
```

/ repeat forever */*
/ display prompt */*
/ input from terminal */*

/ fork off child process */*

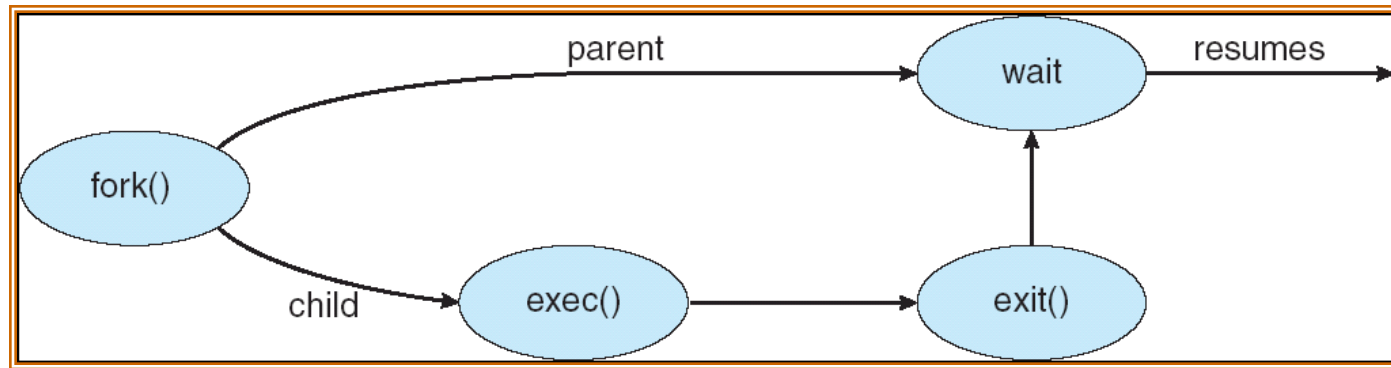
/ wait for child to exit */*

/ execute command */*

exec usata dopo una **fork** rimpiazza lo spazio di memoria con un nuovo programma

wait permette al padre di attendere la terminazione del figlio e quindi, eventualmente, di ricevere risultati dal figlio

... una sintetica rappresentazione ...



Il processo padre, in questo caso, attende la terminazione del processo figlio

... e scenari possibili

- Spazio degli indirizzi

- Il figlio *duplica* quello del padre e non lo modifica
- Il figlio esegue un *nuovo codice*

- Condivisione delle risorse

- Il padre e il figlio condividono *tutte* le risorse
- Il figlio condivide *una parte* delle risorse del padre
- Il padre e il figlio *non* condividono *alcuna* risorsa

- Sincronizzazione

- Vanno in esecuzione *in concorrenza*
- Il *padre attende* la terminazione del *figlio*

Terminazione “normale” di un processo

Il processo esegue l'ultima istruzione e poi chiede a OS di terminare (mediante una istruzione di *exit*)

Tutte le risorse richieste o detenute dal processo vengono deallocate da OS

Terminazione “anormale” di un processo

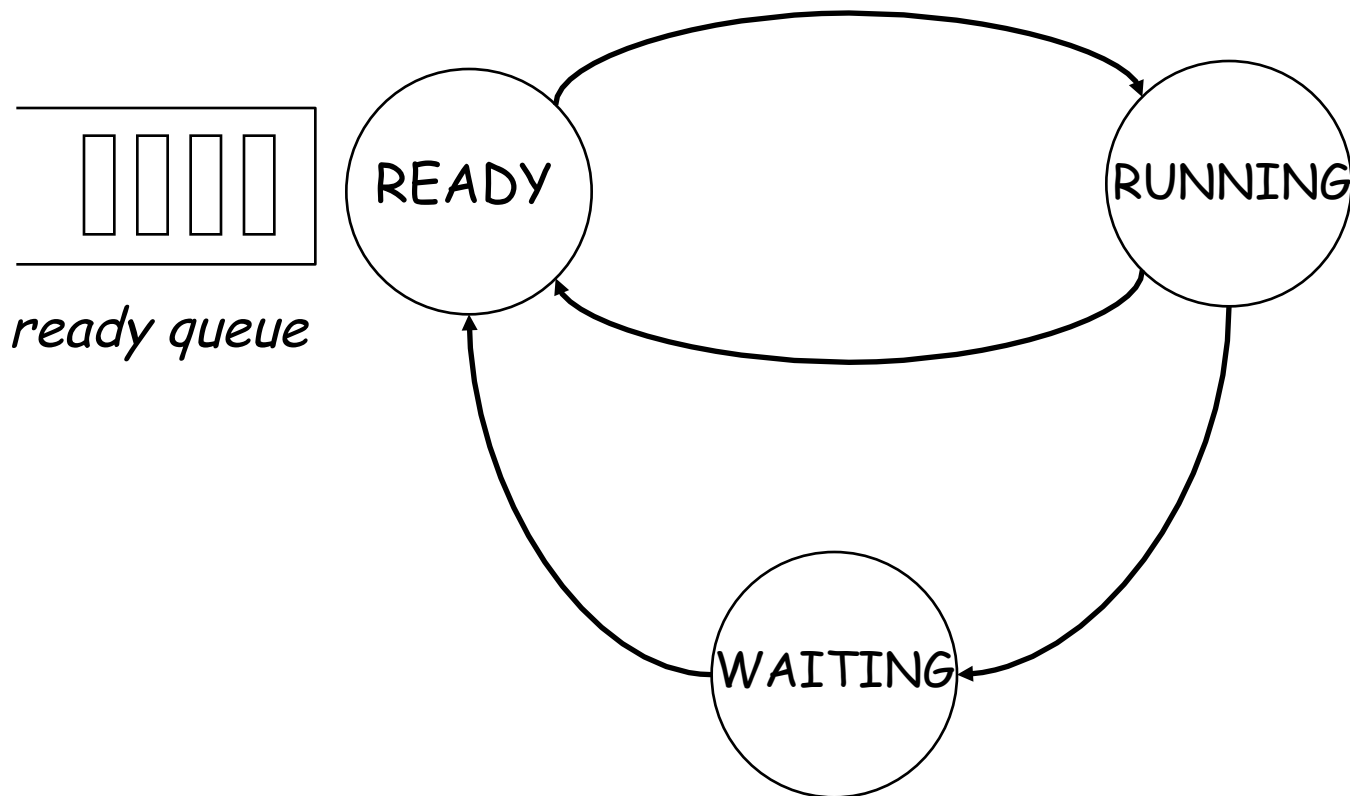
Il padre puo' terminare l' esecuzione di uno dei figli (mediante una istruzione di *abort*)

- perche' -

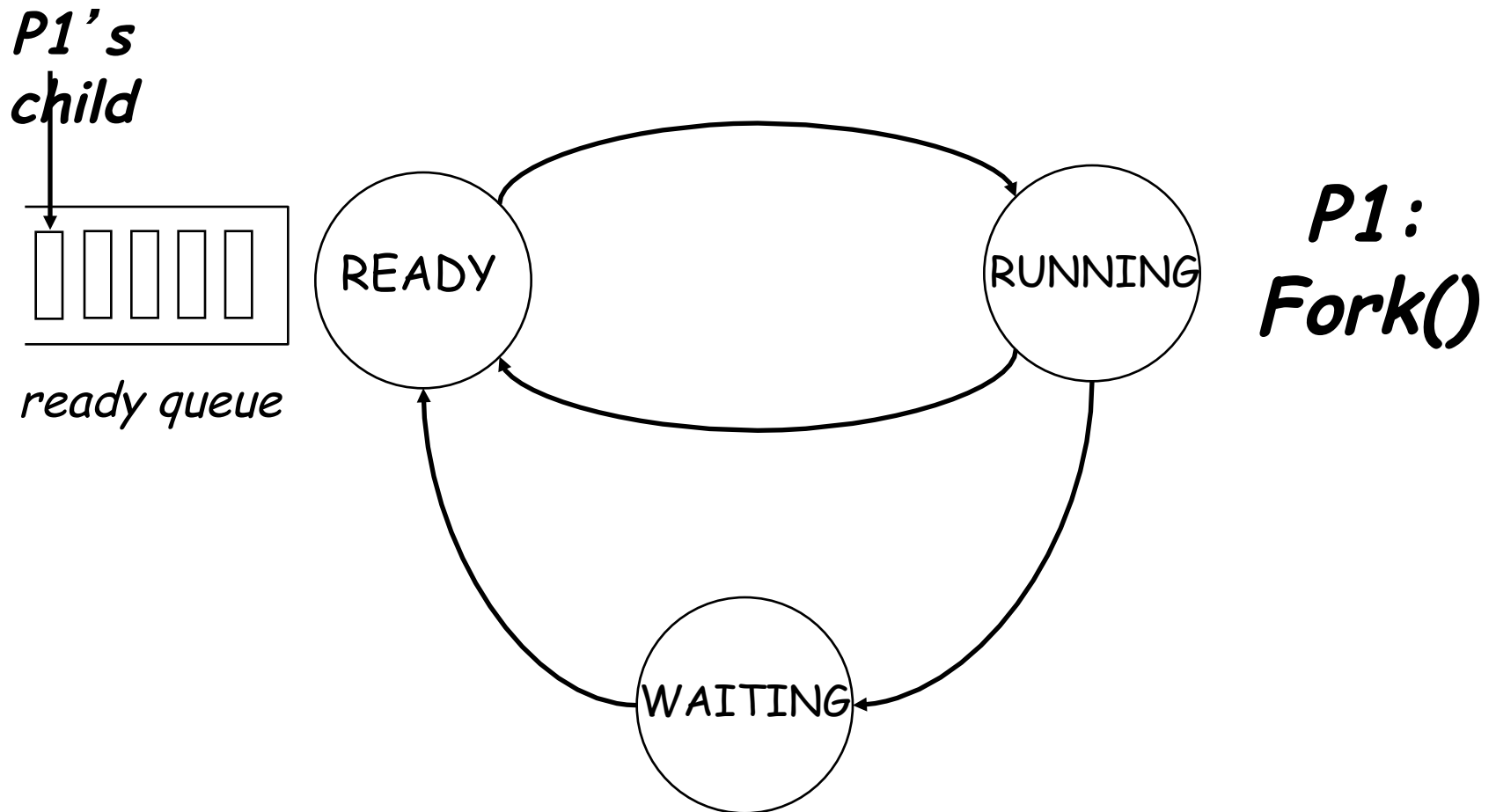


- Il figlio ha ecceduto nell' uso delle risorse (in tal caso il padre deve poter controllare lo stato dei figli)
- Il compito assegnato al figlio non e' piu' richiesto
- Il padre sta terminando (terminazione automatica dei figli in alcuni OS)

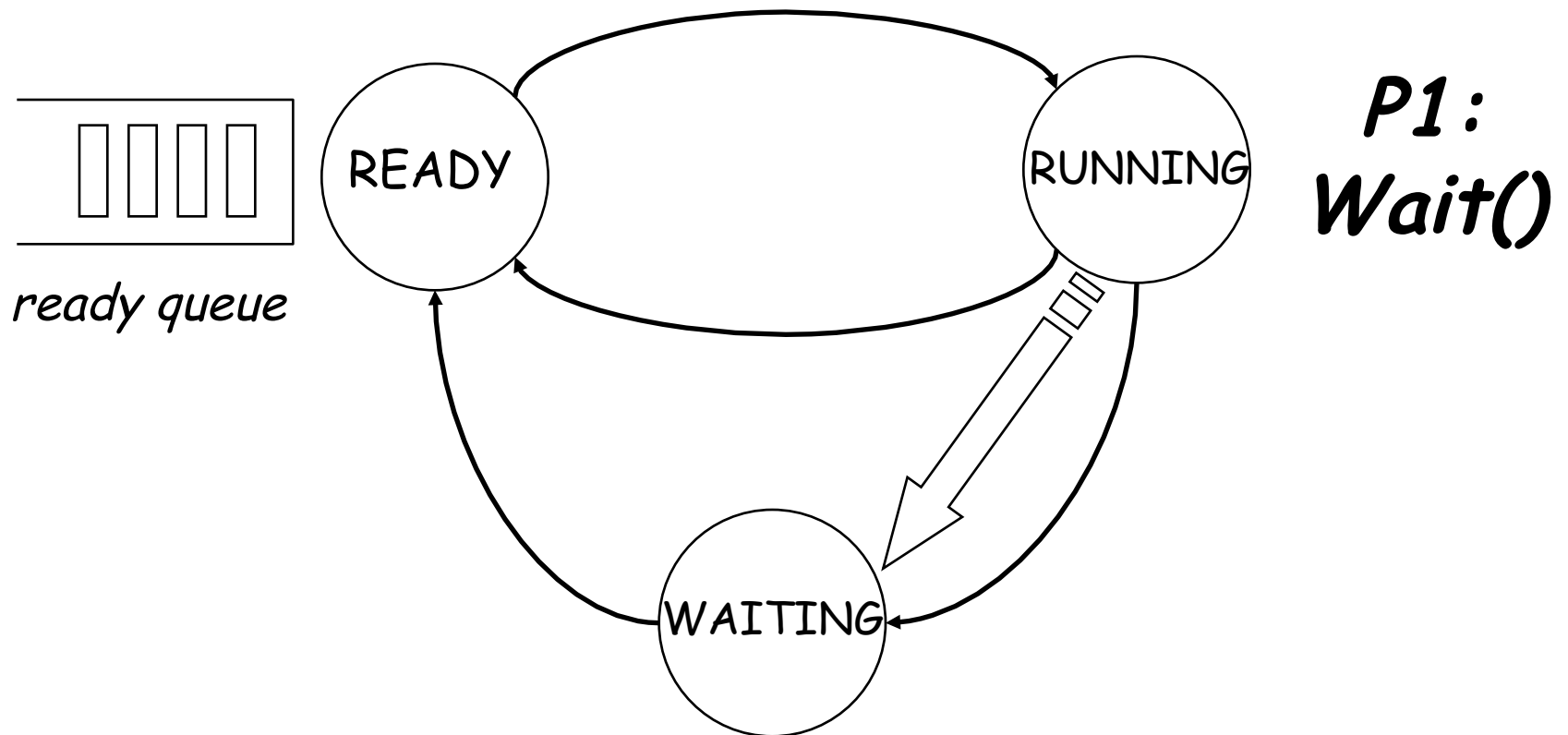
Riassumendo sull'esecuzione di processi...



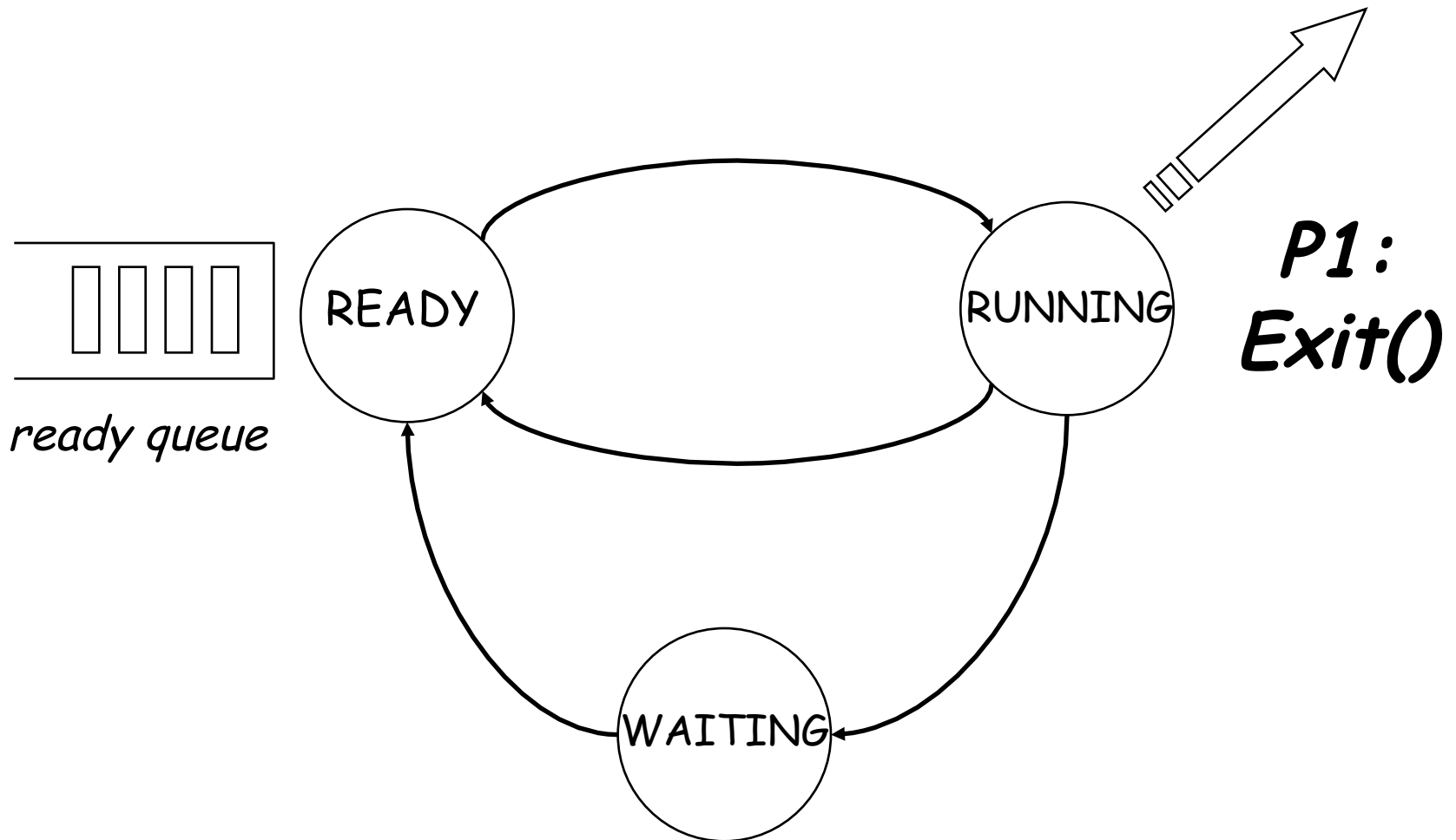
Riassumendo sull'esecuzione di processi...



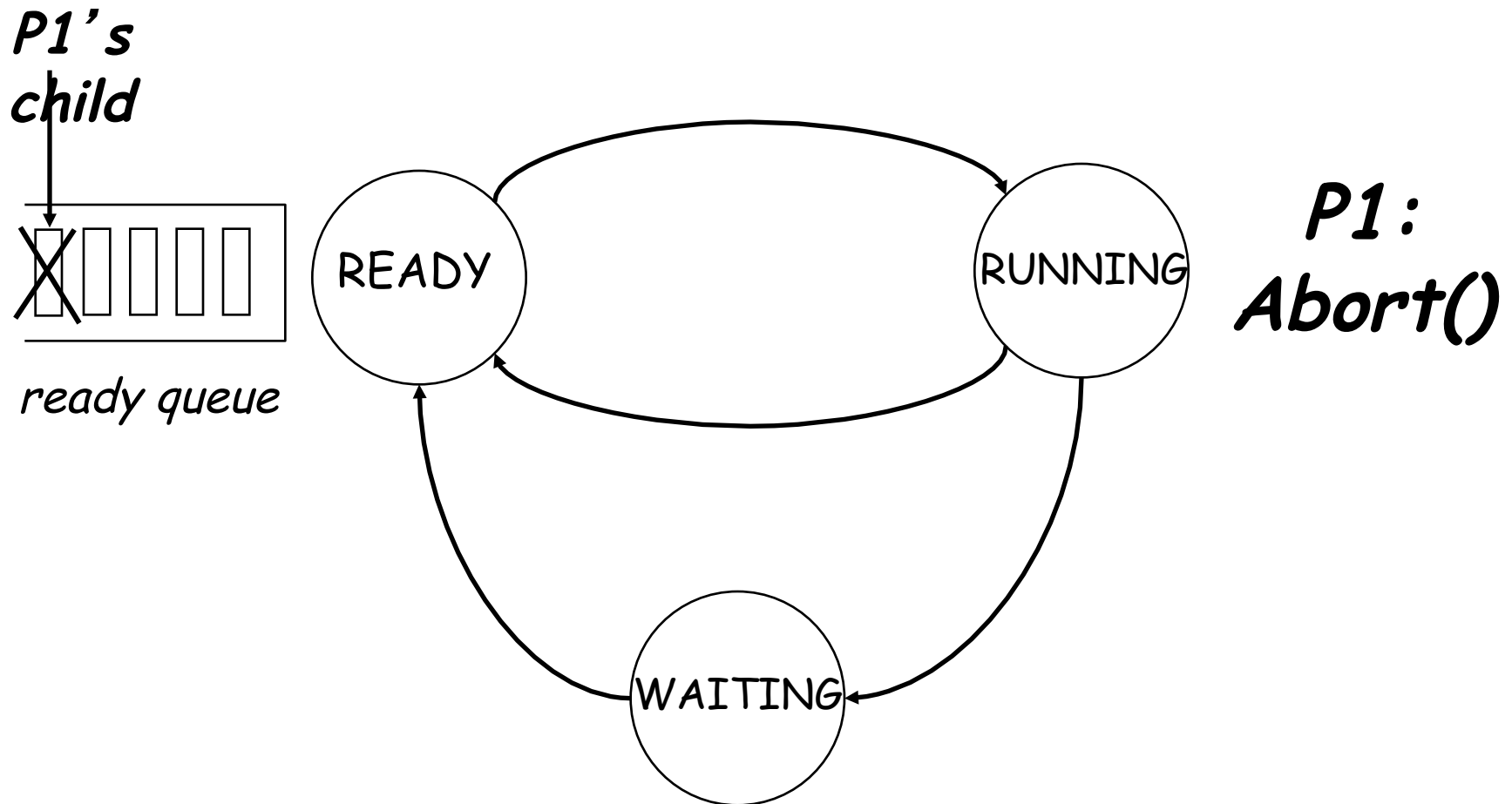
Riassumendo sull'esecuzione di processi...



Riassumendo sull'esecuzione di processi...



Riassumendo sull'esecuzione di processi...



Cooperazione tra processi

I processi possono *influenzare*
l'esecuzione l'uno dell'altro



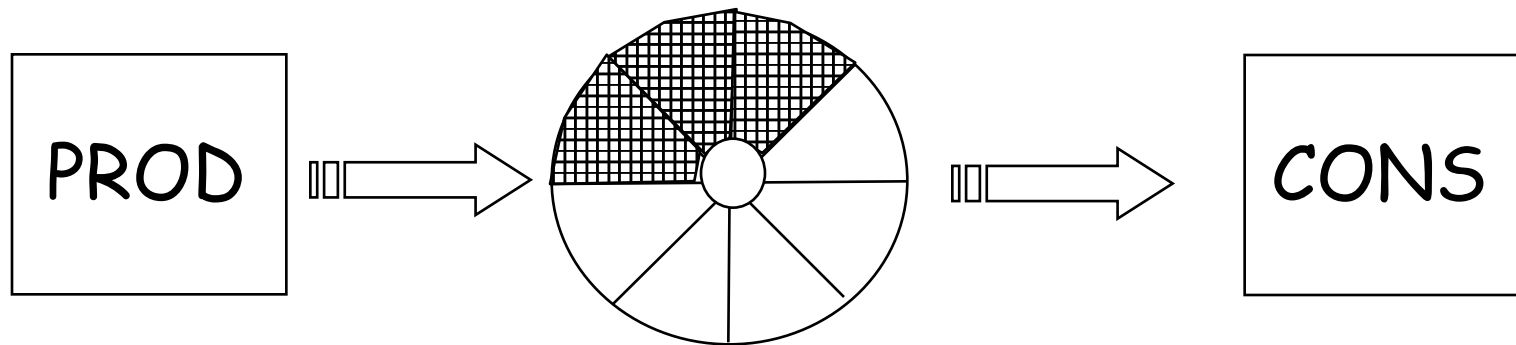
Vantaggi della cooperazione

- Condivisione di informazioni
- Velocizzazione del calcolo
 - Modularita' di compiti
- Convenienza dell'utente

Per regolare questa “influenza” OS ha bisogno di meccanismi di *comunicazione* e di *sincronizzazione*

Una semplificazione del problema della cooperazione: il problema *Produttore-Consumatore*

Il processo *produttore* produce informazioni che sono consumate da un processo *consumatore*



Con un *buffer di taglia finita* bisogna regolamentare le operazioni di produzione e di consumo in maniera “opportuna”

Una soluzione : memoria condivisa

Dichiarazione strutture dati

var n ;

type item = ... ;

*var buffer : array [0... n-1]
of item ;*

in, out: 0... n-1;

Processo produttore

repeat

...

produce an item in *nextp*

...

while $(in + 1) \bmod n = out$
do *no-op*;

$buffer[in] := nextp;$
 $in := (in + 1) \bmod n;$

until *false*;

Processo consumatore

repeat

while $in = out$ do no-op;

$nextc := buffer[out];$

$out := (out + 1) \bmod n;$

...

consume the item in $nextc$

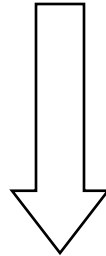
...

until false;

Questione aperta



Con questa soluzione si riescono a utilizzare al piu' $n-1$ locazioni del buffer condiviso



Trovare una soluzione che permetta di utilizzare tutte le n locazioni del buffer condiviso (?!)

Ruolo di OS nella cooperazione

Nella soluzione del Produttore-Consumatore proposta i due processi potevano condividere un buffer e il loro codice era stato *scritto appositamente* per quel tipo di struttura

E' compito di OS mettere in grado, in maniera *trasparente*, i processi di *scambiarsi informazioni tra loro*

Interprocess Communication (IPC)

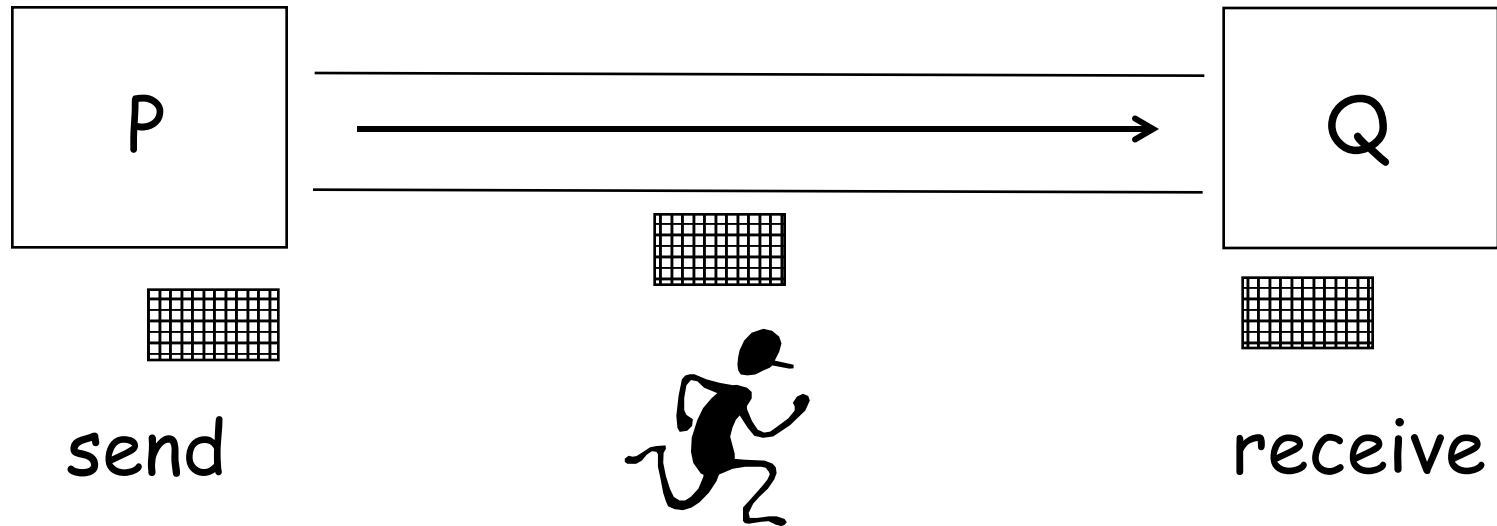
IPC (parte di OS destinata a gestire la comunicazione) mette a disposizione dei processi utente essenzialmente due system calls :

- *send*
- *receive*

Cosa fare per comunicare ?!

Se i processi P e Q vogliono *comunicare* devono:

- stabilire un *canale di comunicazione* tra essi
- scambiare messaggi usando *send* e *receive*



Ad un *canale logico* corrisponde un *canale fisico* che puo' essere o una cella di memoria condivisa oppure un bus hardware

Questioni implementative...

- Come si stabilisce una connessione?
- Può *un canale* essere associato a *più di due processi*?
- Viceversa, *quanti canali* possono essere associati *ad un'unica coppia* di processi?
- Un canale è uni- o bi-direzionale?

... le risposte dipendono dagli
attributi principali di una
comunicazione:

A. diretta/indiretta

B. buffering/no buffering

C. sincrona/asincrona

A. Comunicazione diretta

Ognuno dei due processi deve conoscere il nome dell'altro processo con il quale vuole comunicare

send (*Q*, *message*) - manda *message* al processo *Q*
receive(*P*, *buffer*) - riceve in *buffer* dal processo *P*



contenuto

contenitore

Proprieta'

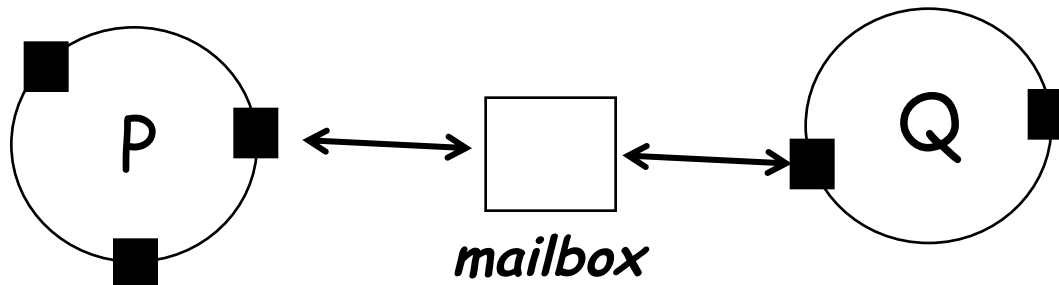
- La connessione e' stabilita in maniera automatica
- Un canale e' fissato tra due processi
- Fra due processi ci puo' essere un unico canale
- Il canale puo' essere bidirezionale

Comunicazione indiretta...

I messaggi sono mandati e ricevuti attraverso *mailboxes* (associati a *porte*)

Ogni mailbox ha un *identificatore unico*, noto ai processi che ne vogliono fare uso per comunicare

I processi possono comunicare solo
se condividono un mailbox



... operazioni , proprieta' ...

Operazioni

Crea un mailbox

send e receive attraverso il mailbox

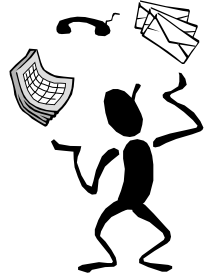
Distruggi il mailbox

Proprieta'

- La connessione e' stabilita solo se i processi condividono il mailbox
- Un mailbox puo' essere associato a piu' processi
- Fra due processi ci puo' essere piu' di un mailbox
- Il mailbox puo' funzionare in modo bidirezionale

... ed un problema

*P1, P2, and P3 condividono mailbox A
P1 manda messaggi; P2 e P3 ricevono*



Chi riceverà' effettivamente i messaggi in arrivo?

Possibili soluzioni

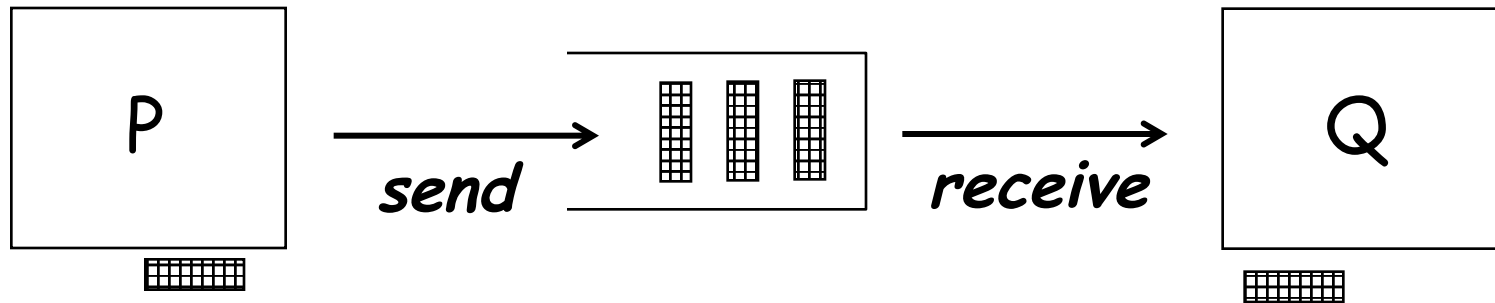
- Permettere che un mailbox sia associato a soli due processi
- Permettere che solo un processo alla volta possa eseguire una *receive*
- Delegare OS a selezionare arbitrariamente il ricevente ogni volta (al mandante verrà notificato dopo il nome di chi ha ricevuto)
- Multicasting e Broadcasting

B. Buffering / No buffering...

Un canale con buffering permette di associare una *coda di messaggi* ad esso

Questo significa che un processo che vuole inviare un messaggio lo può fare *virtualmente* se il canale e' gia' occupato

... e tre implementazioni possibili



Capacita' **nulla** - rendezvous (no buffering)

Capacita' **finita** - il sender aspetta solo se il canale e' pieno

Capacita' **infinita** - la comunicazione non e' mai ritardata

C. Sincrona / Asincrona...

Una operazione di *receive* e' *sempre bloccante*
(*almeno per un timeout*)

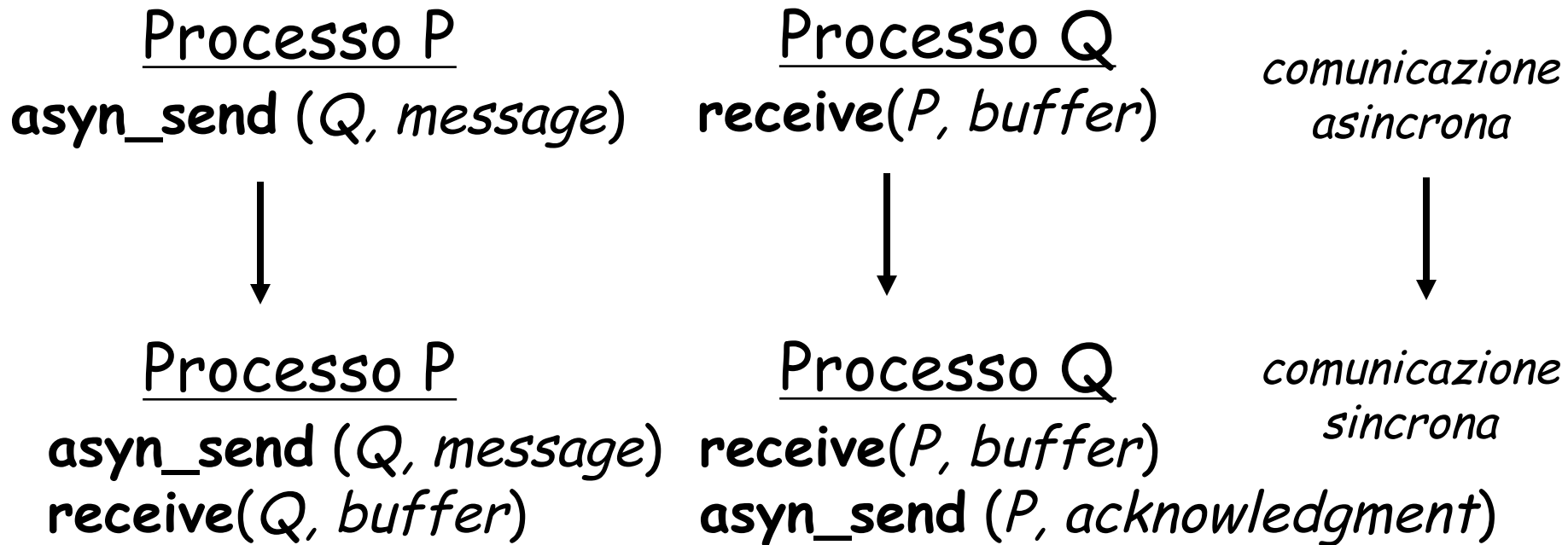
Una operazione di *send* *puo' essere bloccante*

Comunicazione sincrona

Il processo che esegue la *send* si blocca
attendendo che la corrispondente *receive*
venga eseguita prima di proseguire

Un canale con *buffer nullo* *puo' ospitare*
solo comunicazioni *sincrone*

...come utilizzare primitive asincrone per costruire una comunicazione sincrona



In alcuni OS l'operazione di *send* attende una *reply* entro un certo *timeout*

Eccezioni da gestire per la comunicazione



- Un processo ricevente attende un messaggio da un processo che ha terminato
- Un processo invia un messaggio ad un processo che ha terminato (caso buffering e no buffering)
- Un messaggio inviato da un processo ad un altro viene perduto (timeout)
- Un messaggio inviato da un processo ad un altro viene danneggiato (il ricevente puo' notificarlo al mandante, se e' in grado di riconoscere l'errore)

THREADS

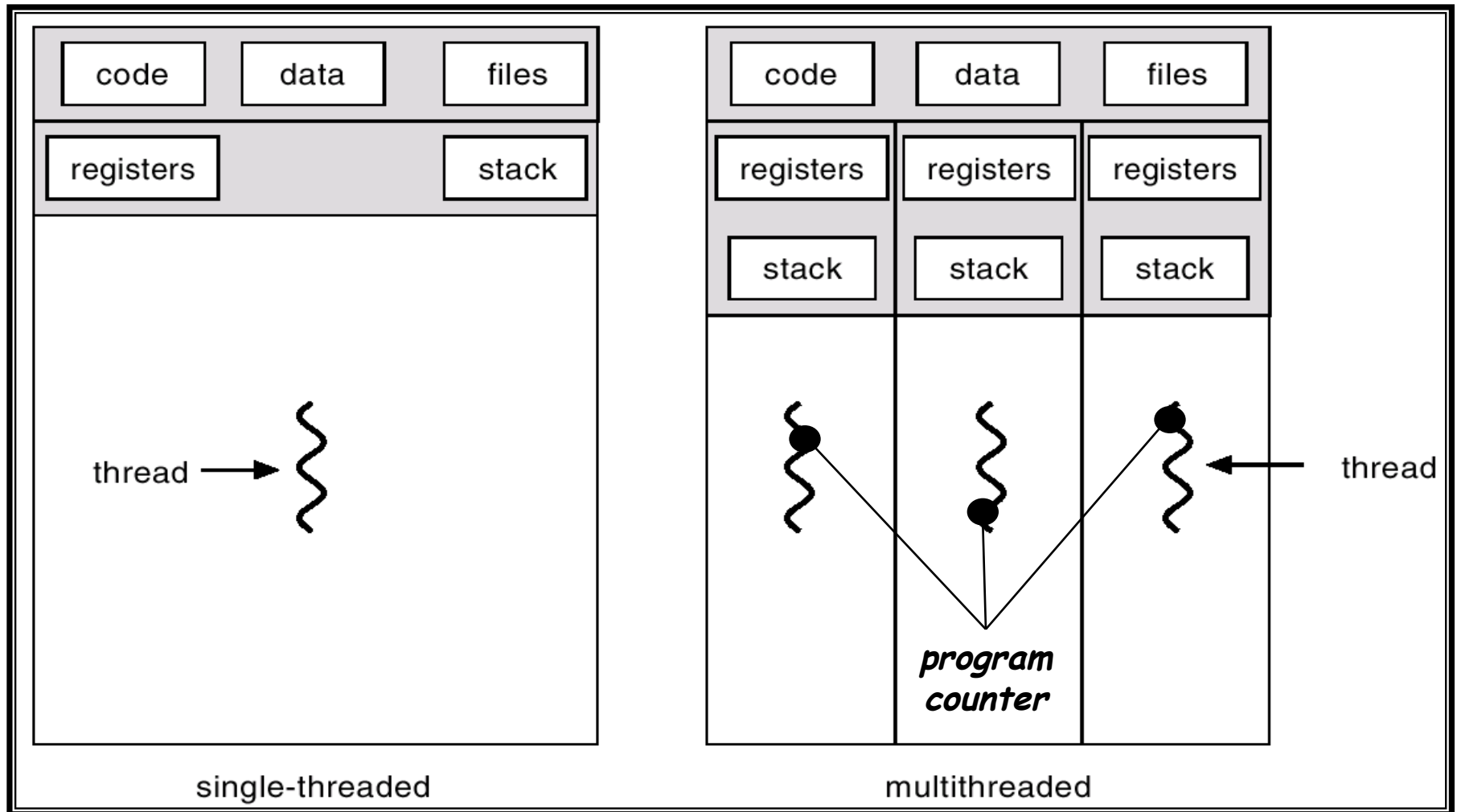
Threads

Una *thread*, o *processo leggero*, e' l'unita' di base dell'utilizzazione della CPU

Item <i>propri</i> di una thread	→	<i>Stato</i> <i>Program counter</i> <i>Insieme di registri</i> <i>Stack</i>
Item condivisi con le altre thread ad essa associate (a formare un <i>task</i>)	→	<i>Sezione di codice</i> <i>Sezione dati</i> <i>Risorse di OS</i>

Un *processo pesante* (tradizionale) e' uguale ad un task con un'unica thread

Multiple threads all' interno di un task

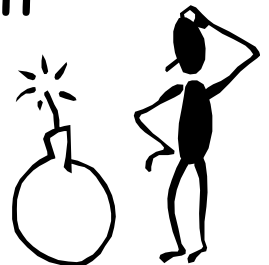


Alcune proprietà delle thread...

- *Una sola thread alla volta in esecuzione*
- Una thread può *leggere* o *scrivere nello spazio* di qualunque altra thread dello stesso task
- In caso di *bloccaggio* di una thread, *un'altra* dello stesso task in molti casi *potrebbe continuare* la sua esecuzione, e quindi il processo non sarebbe bloccato
- Il *context-switching* a livello di thread è più rapido in quanto non coinvolge un uso pesante della memoria

... ed alcune considerazioni

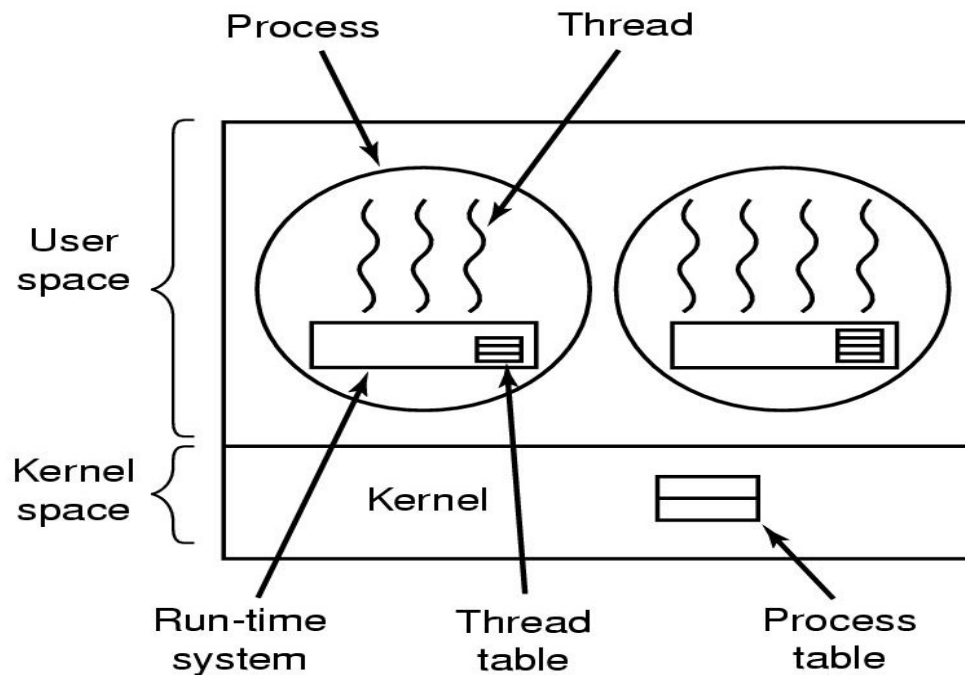
- La cooperazione di multiple thread conferisce un piu' *alto throughput* e migliori performance
- Applicazioni che devono *condividere risorse* si giovano di thread, ex. un task che deve fornire dati a piu' macchine remote in un file system di rete (es. un *web server*)
- Esse permettono di *introdurre parallelismo* nella esecuzione di task che eseguono system call bloccanti



Threads: gestione a livello utente

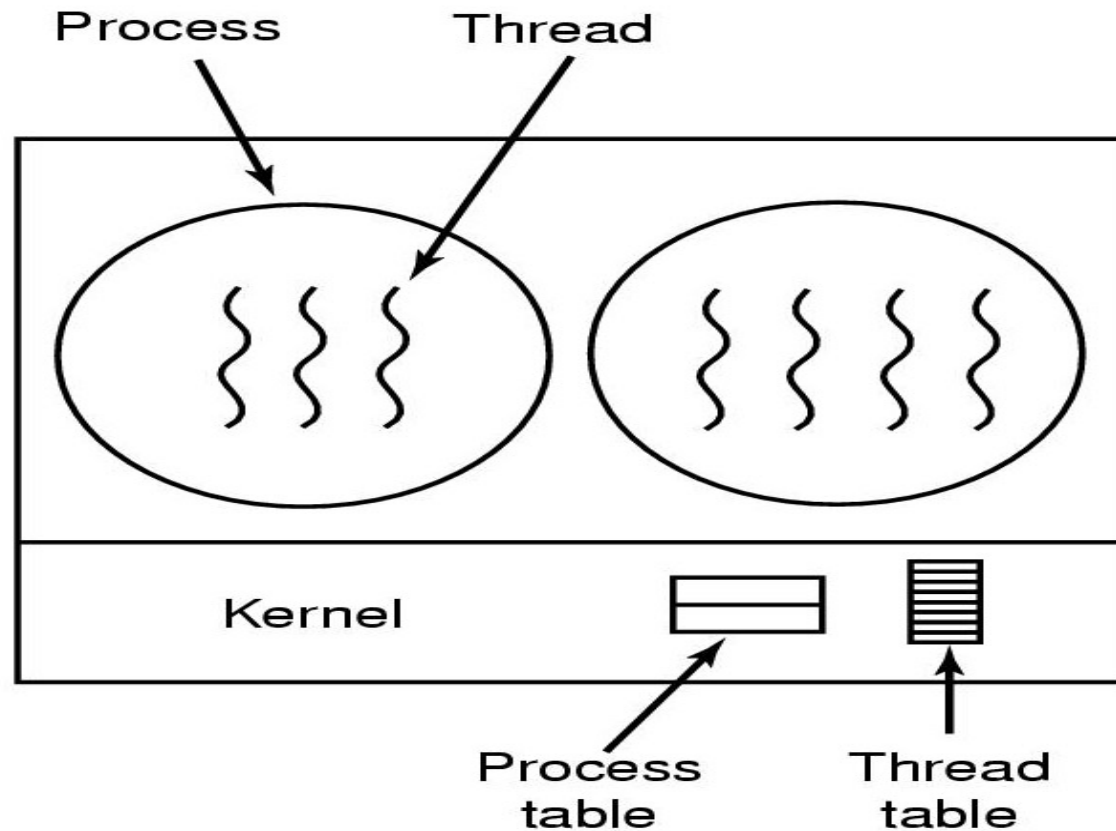
Un insieme di funzioni di libreria per gestire le thread

Il kernel allora vede ogni task come un singolo processo, le thread non sono unita' visibili da



- Passaggio veloce da una thread all'altra
- Bloccata una thread bloccato l'intero task
- Sbilanciamento nello scheduling

Threads: gestione a livello kernel



Un compito in piu' per OS :
gestire le threads
(scheduling, creazione, etc.)