# Sistemi Operativi

Laurea in Ingegneria Informatica
Università Roma Tre
Docente: Romolo Marotta

# Processi e Threads

1. Processi
2. Rappresentazione di processi
3. Strutture dati e scheduling
4. Process e mode switch
5. Processi in sistemi UNIX
6. Thread e multi-threading
7. Thread in sistemi UNIX

# Cos’è un processo?

- Un programma in esecuzione?
- Un’istanza di programma in esecuzione
- 1 programma => 0..N processi
- 1 processo => 1 programma (per ora)

# Elementi caratterizzanti di un processo

- Un processo è associato a:
  - Un programma
  - I dati su cui opera
  - Almeno uno stack
  - Dati di contesto (contenuti nei registri di processore)
  - Le risorse hardware di cui ha richiesto l'accesso
  - Un identificativo
  - Statistiche
  - Uno stato (e.g., running)

Process Control Block (PCB)

Immagine di processo

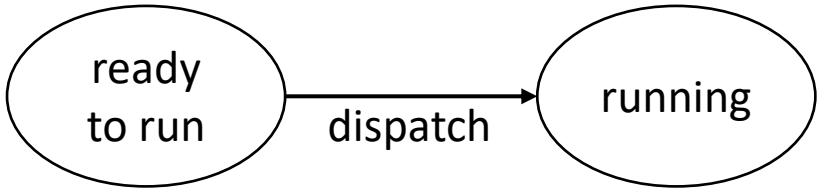
# Modello di esecuzione di un processo

- Un modello è **un'astrazione** di un generico sistema il cui scopo è comprenderne il comportamento del sistema modellato
  - Tralasciare i dettagli non rilevanti per gli obiettivi del modello
- Dato un modello è possibile:
  - Tradurlo in codice
  - Utilizzarlo per prendere decisioni
- Il sistema operativo utilizza un modello di esecuzione per caratterizzare il comportamento di un processo
  - Una macchina a stati
    - Nodi
    - transizioni

# Modello di esecuzione di un processo

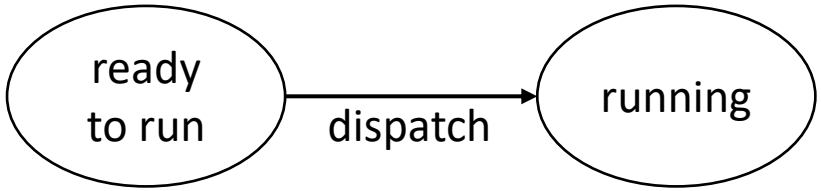
- Esempio di stati in un sistema con uniprogrammazione

| **Proc.** | Run Pro A | I/O | Run Pro A | I/O | Run Pro B | I/O | Run Pro B | I/O | Run Pro C | I/O |
|---|---|---|---|---|---|---|---|---|---|---|



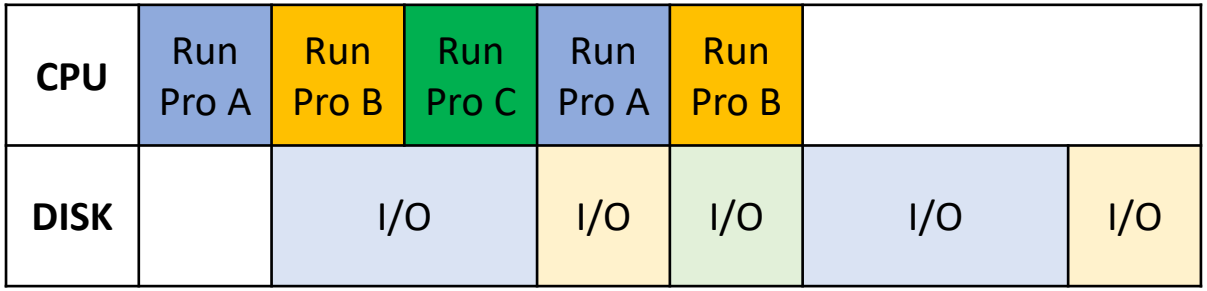
# Modello di esecuzione di un processo

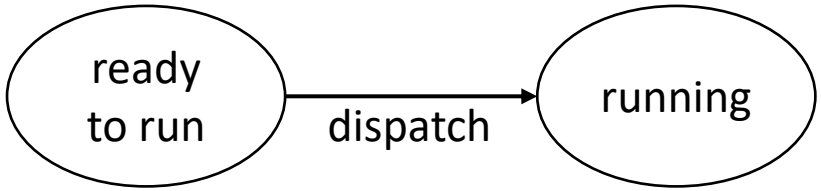
- Esempio di stati in un sistema con multiprogrammazione

# Modello di esecuzione di un processo

- Esempio di stati in un sistema con multiprogrammazione

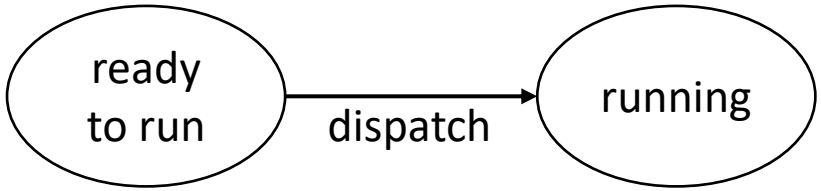


# Modello di esecuzione di un processo

- Esempio di stati in un sistema con multiprogrammazione ➡ Time-out

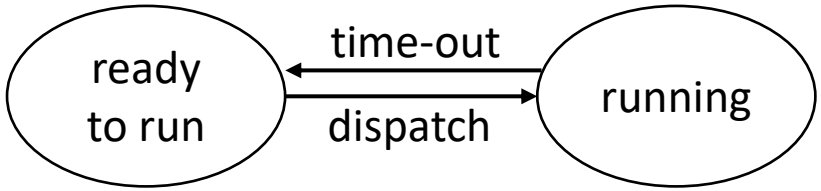
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **CPU** | Run Pro A | Run Pro B | Run Pro C | Run Pro A | Run Pro B | | |
| **DISK** | | I/O | | I/O | I/O | I/O | I/O |



# Modello di esecuzione di un processo

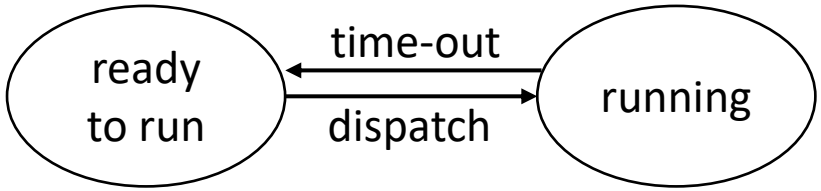
- Esempio di stati in un sistema con multiprogrammazione ➡ Time-out

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **CPU** | Run Pro A | Run Pro B | Run Pro C | Run Pro A | Run Pro B | | |
| **DISK** | | I/O | | I/O | I/O | I/O | I/O |

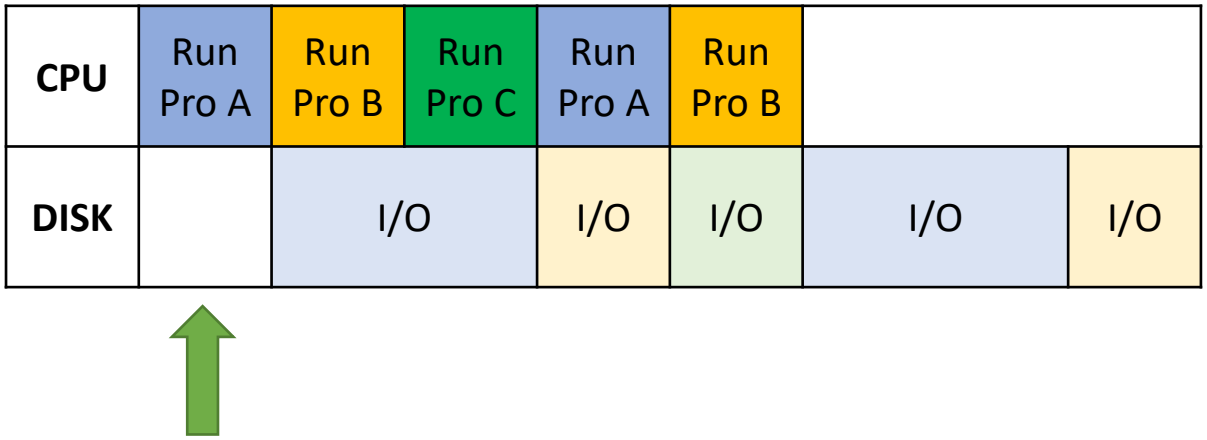


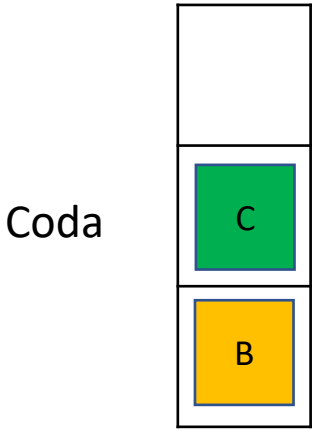
# Modello di esecuzione di un processo

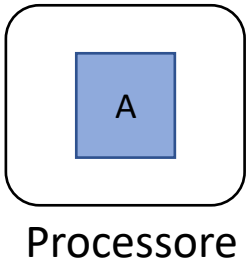
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **CPU** | Run Pro A | Run Pro B | Run Pro C | Run Pro A | Run Pro B | | |
| **DISK** | | I/O | | I/O | I/O | I/O | I/O |

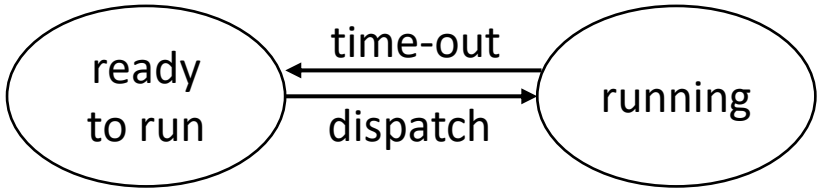


# Modello di esecuzione di un processo

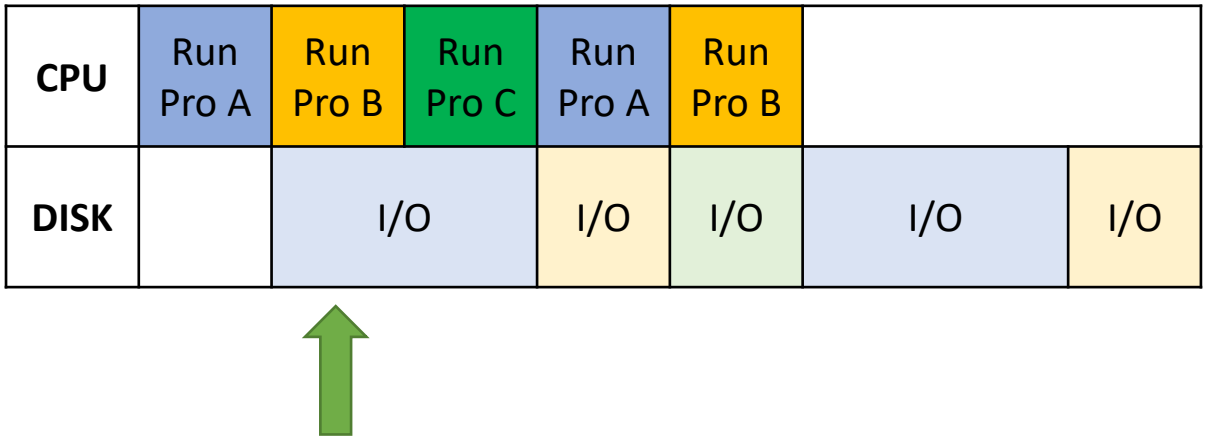


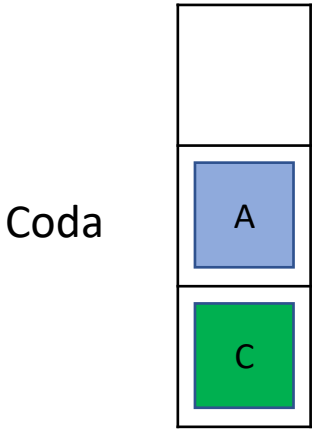



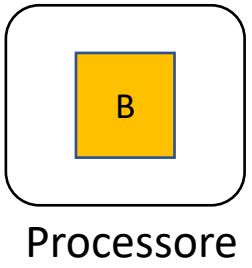



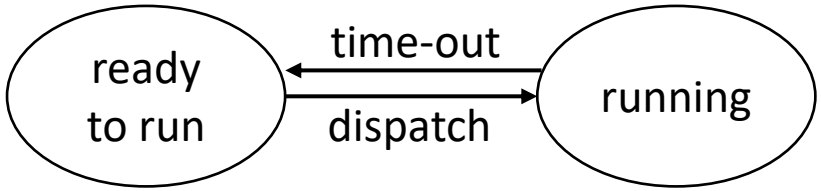



# Modello di esecuzione di un processo

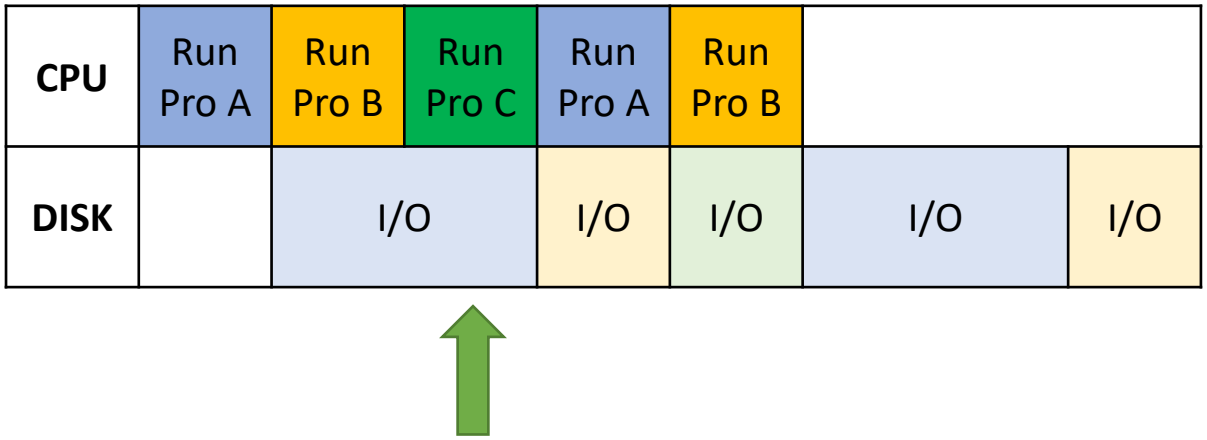


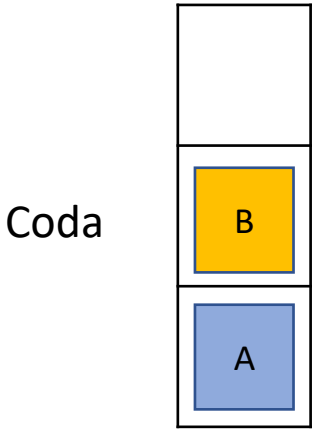



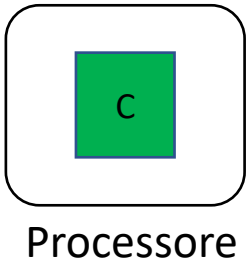



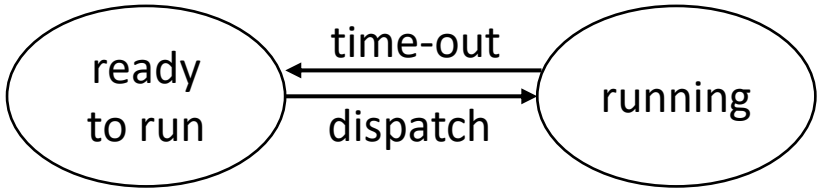



# Modello di esecuzione di un processo

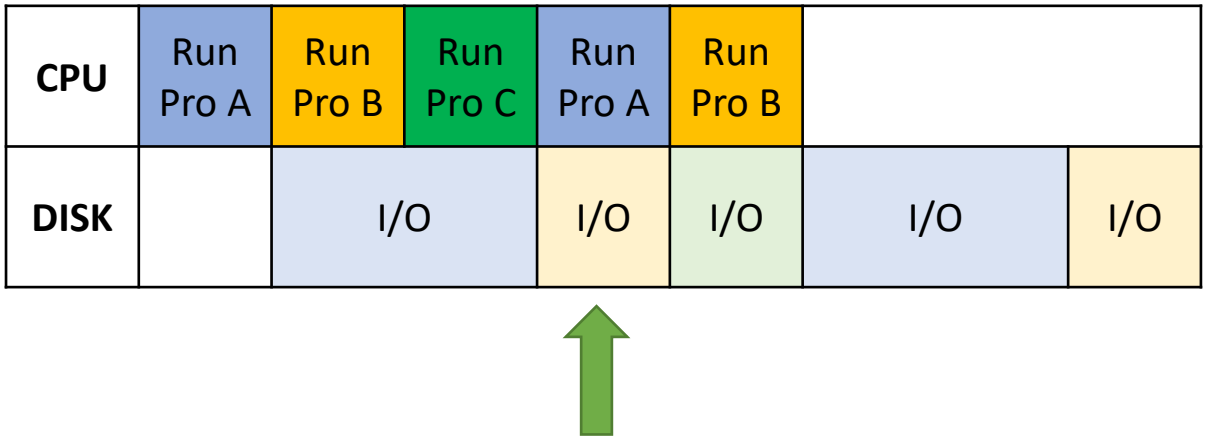


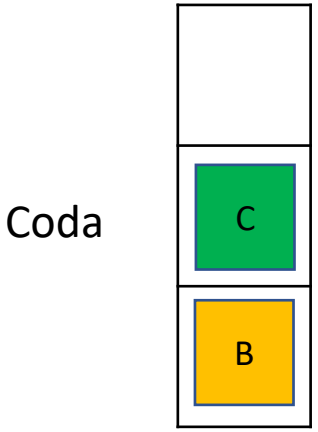



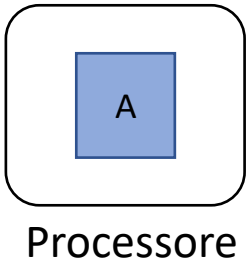



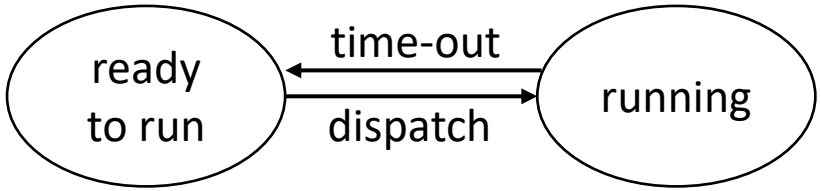



# Modello di esecuzione di un processo

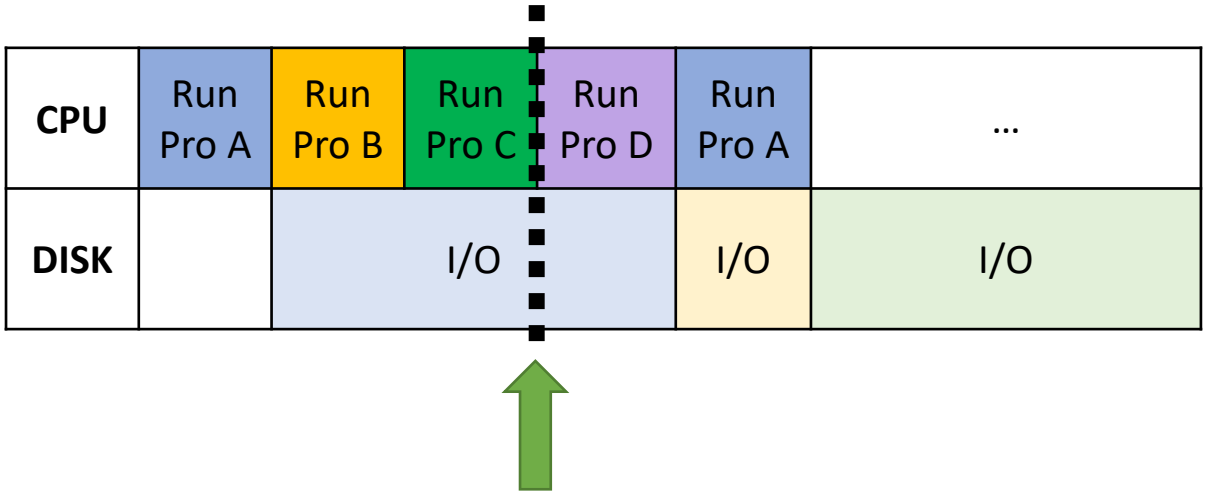


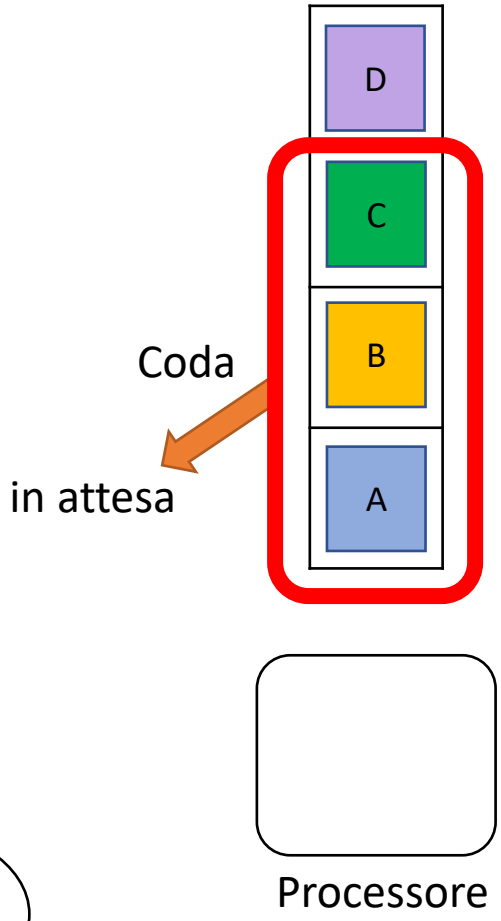



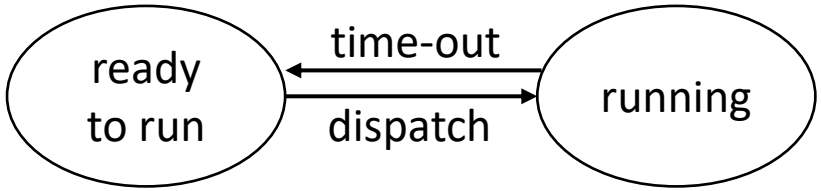



# Modello di esecuzione di un processo

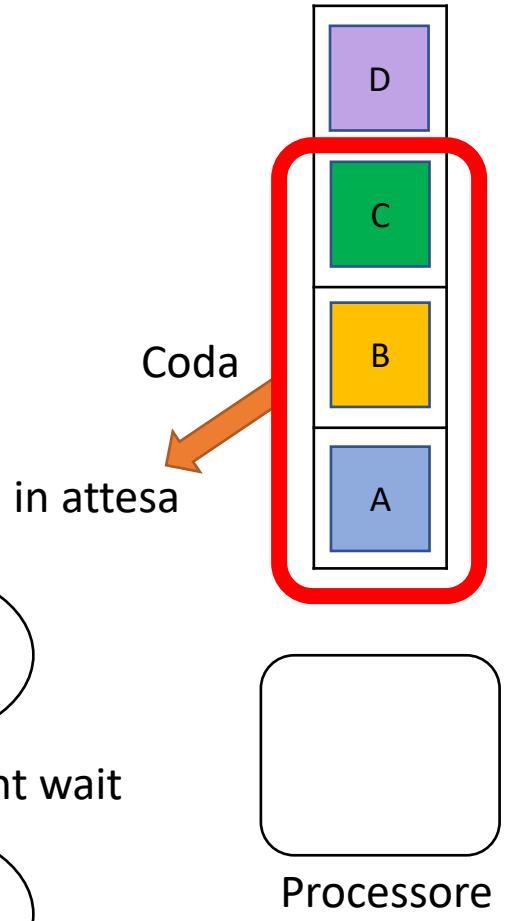


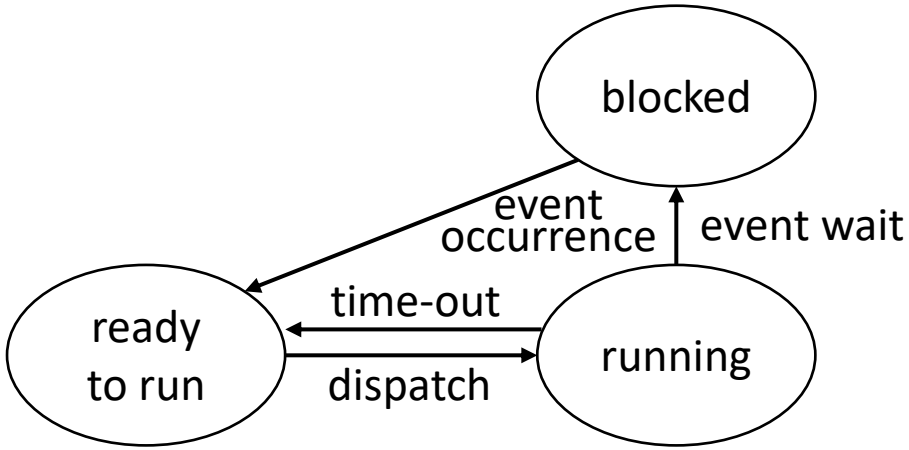



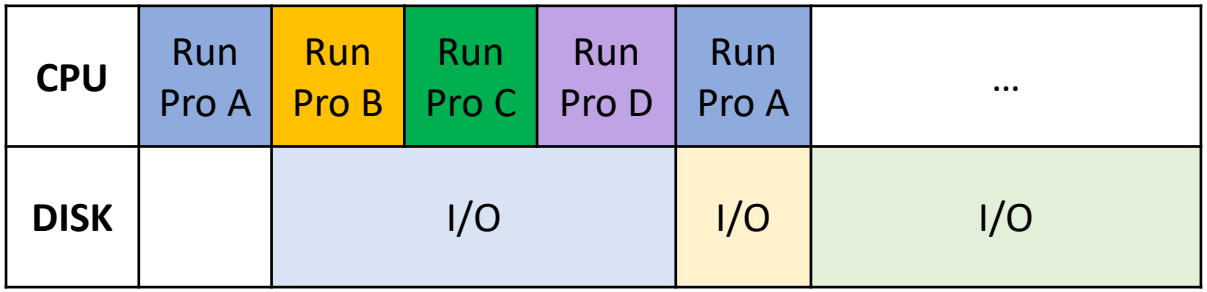
# Modello di esecuzione di un processo

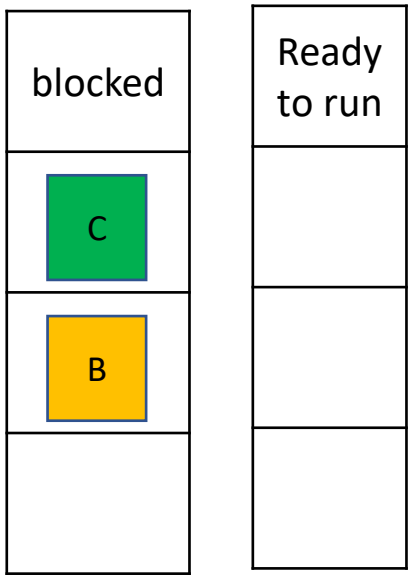
| | | | | | | |
|---|---|---|---|---|---|---|
| **CPU** | Run Pro A | Run Pro B | Run Pro C | Run Pro D | Run Pro A | … |
| **DISK** | | I/O | | | I/O | I/O |

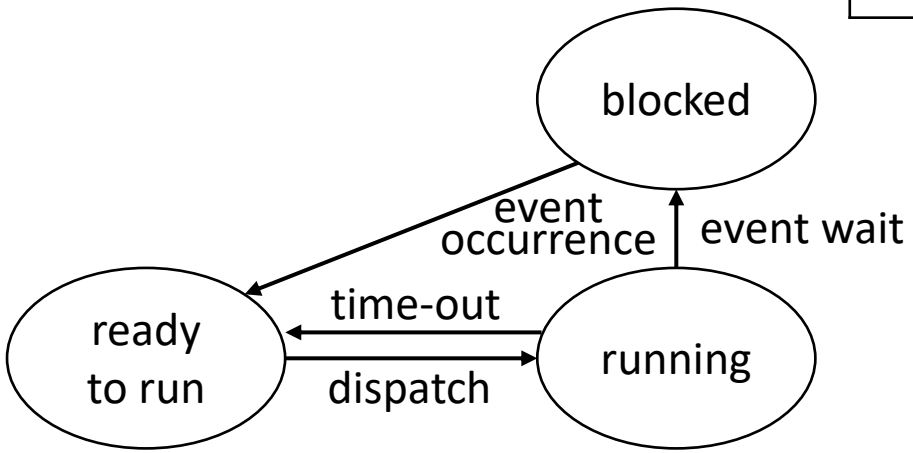


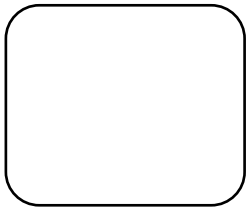



# Modello di esecuzione di un processo

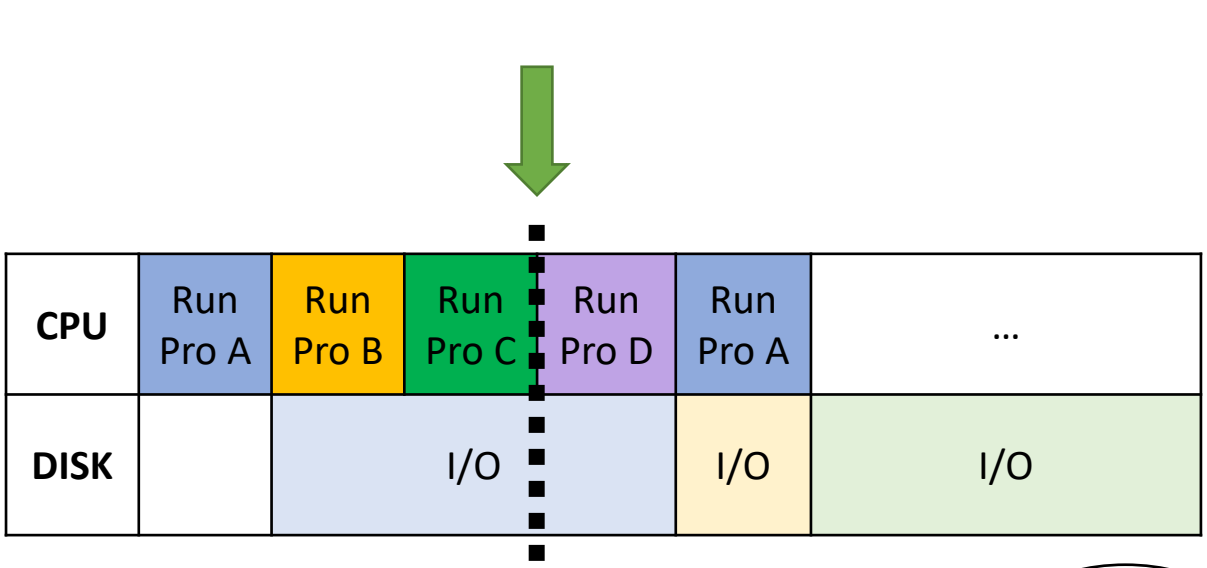


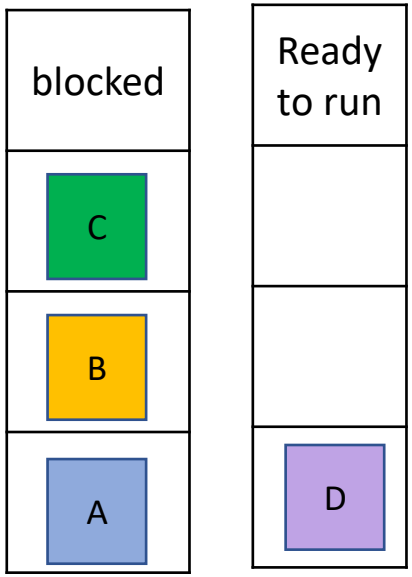



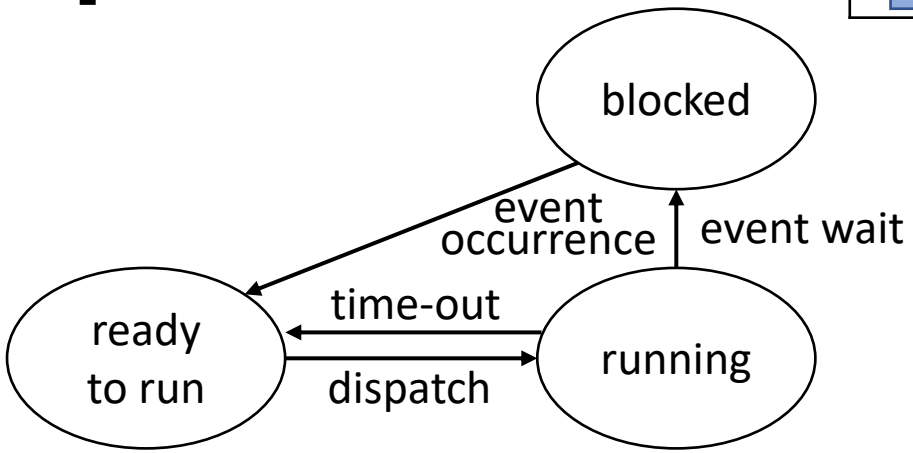



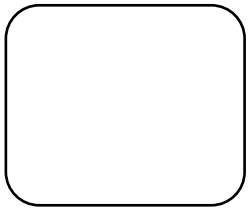


Processore

# Modello di esecuzione di un processo

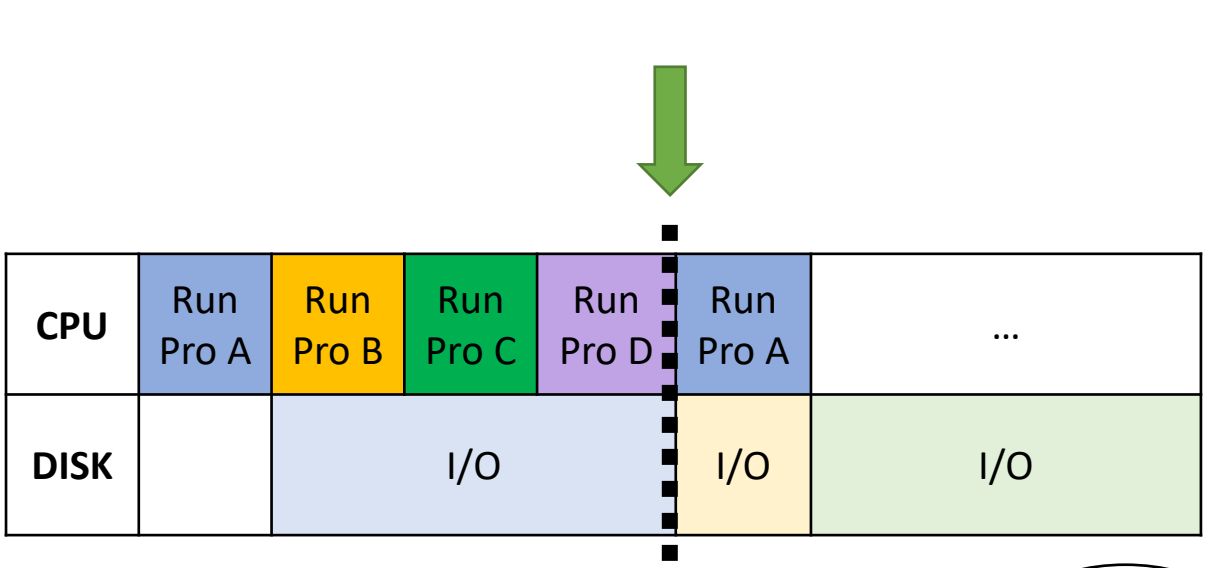


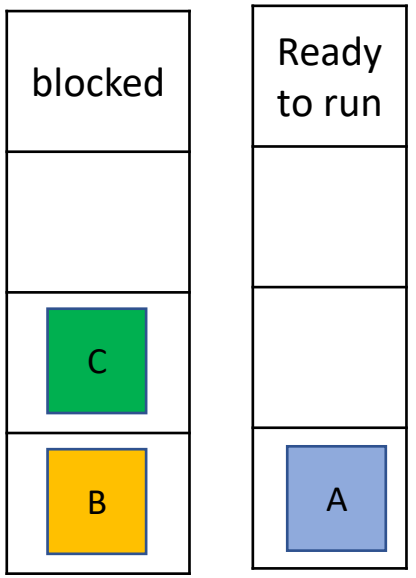



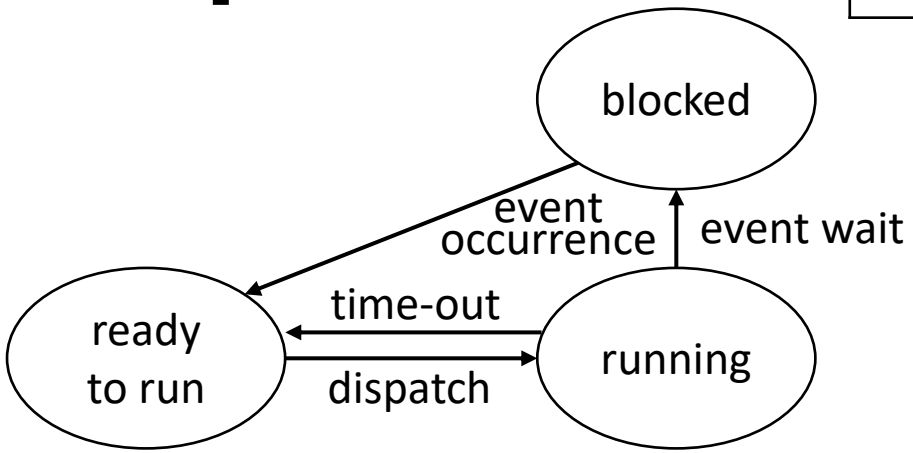



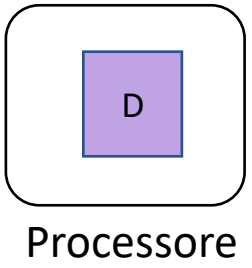


Processore

# Modello di esecuzione di un processo

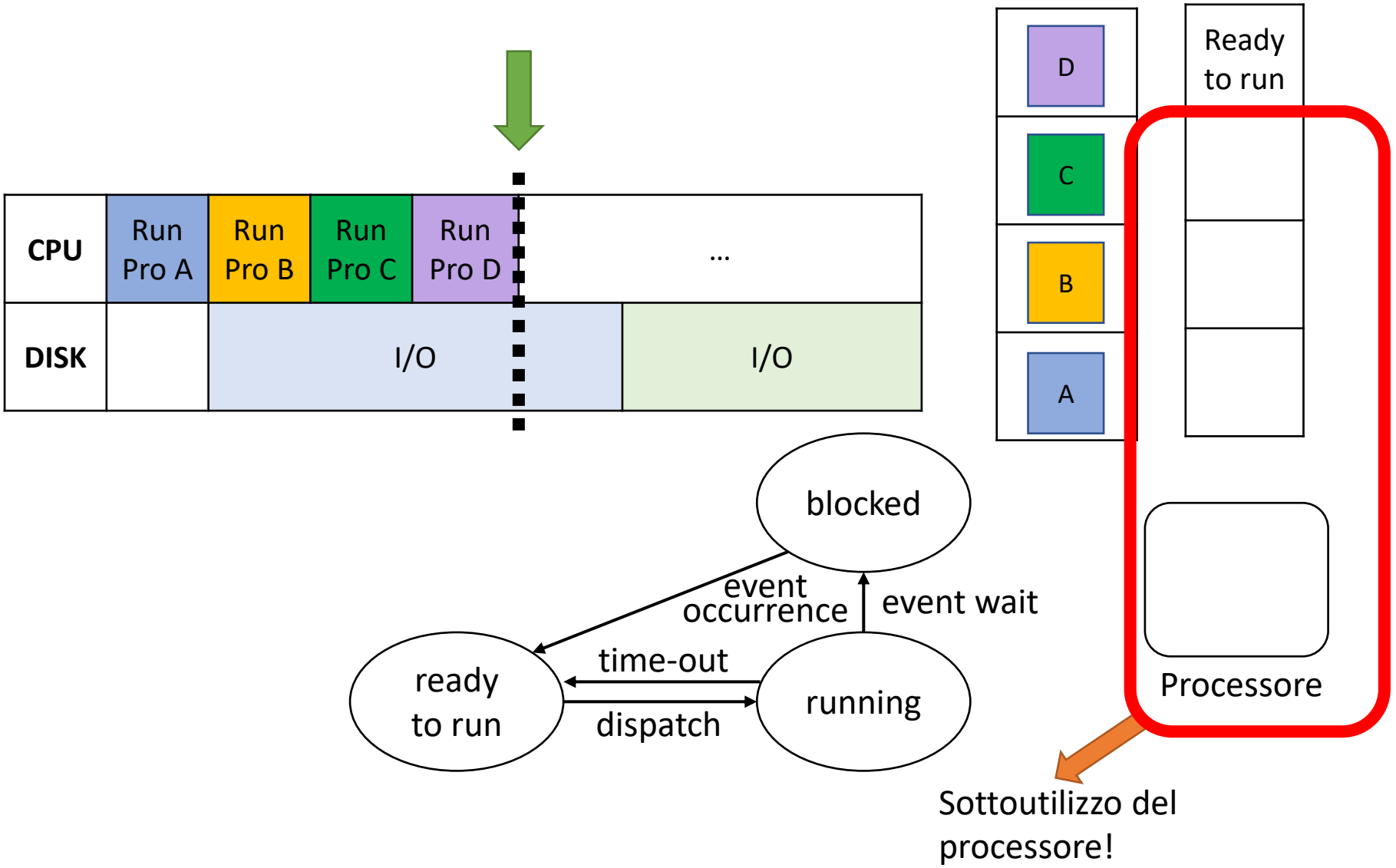


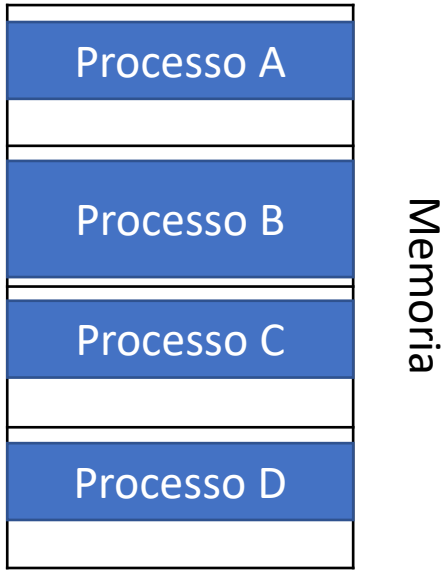



# Modello di esecuzione di un processo



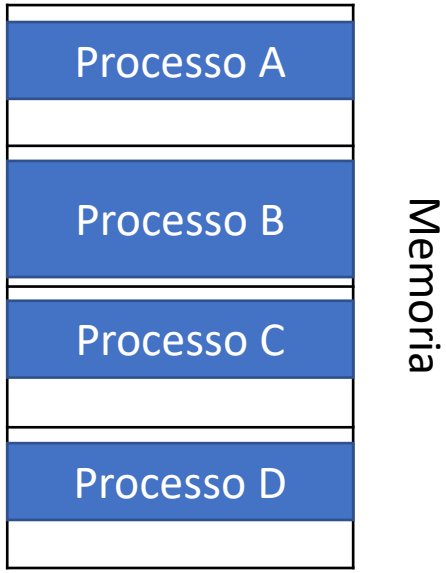
# Osservazione

- Il processore è molto più veloce di dispositivi di I/O
- La multiprogrammazione aiuta ad evitare il sottoutilizzo del processore
- **È necessario avere il maggior numero possibile di processi ready-to-run**
- Qual è il livello massimo di multiprogrammazione raggiungibile?
  - Il numero di processi che la capacità di memoria riesce contenere



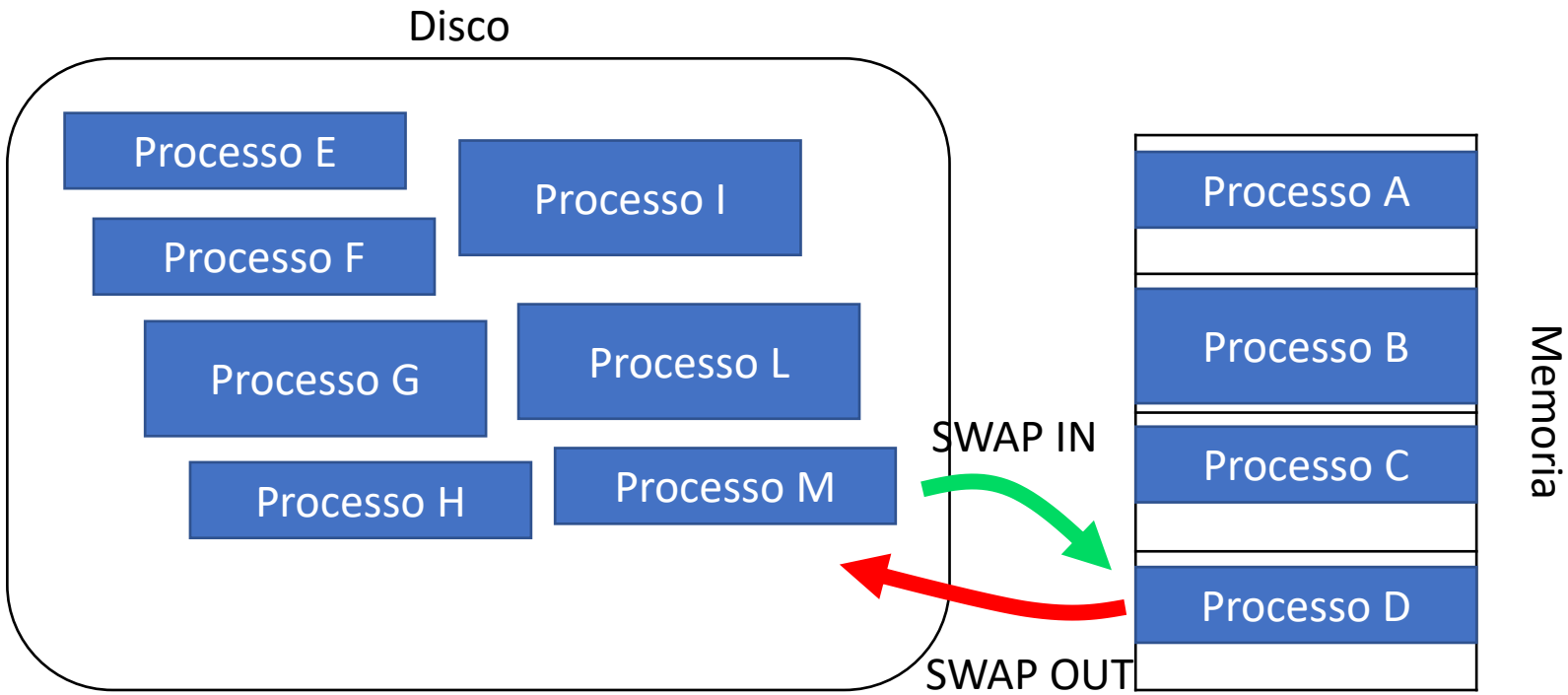
# Osservazione

- Come aumentare il livello di multiprogrammazione?
  - Aumentare la capacità della memoria
- Problemi:
  - la memoria è costosa
  - fissata la quantità di memoria il sottoutilizzo del processore è ancora possibile

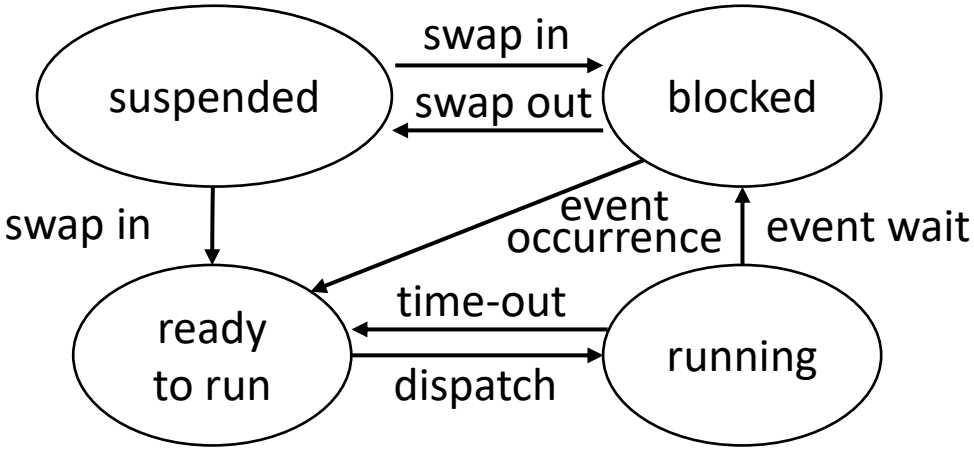


# Osservazione

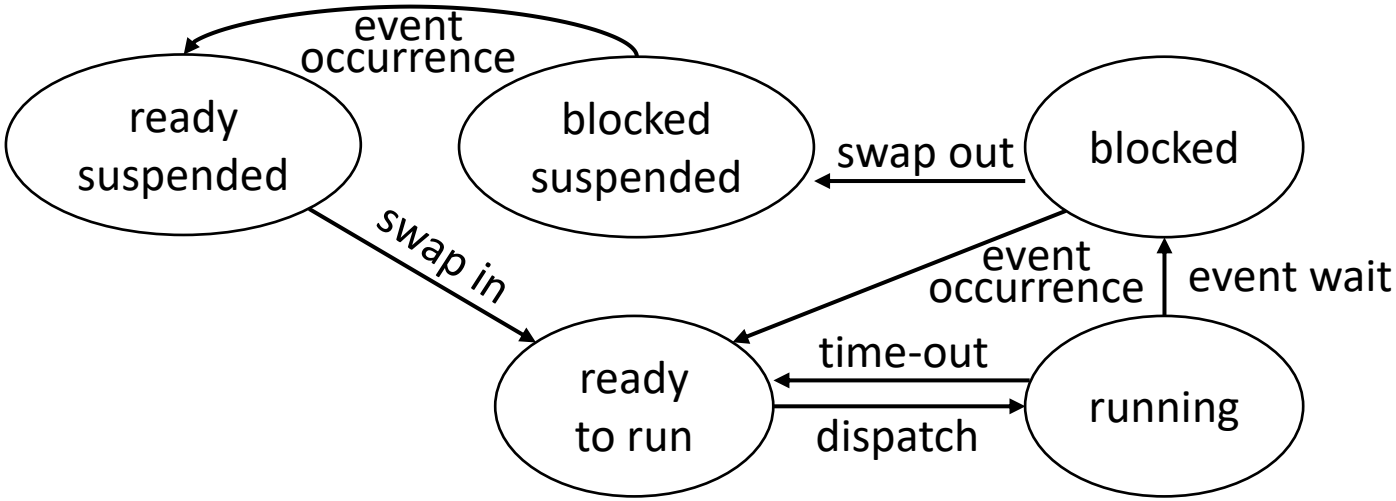
- Come aumentare il livello di multiprogrammazione?
  - Aumentare la capacità della memoria
  - Utilizzare dispositivi di storage per rimuovere processi dalla memoria
    - Il costo per bit è minore rispetto alle memorie (e.g. RAM)



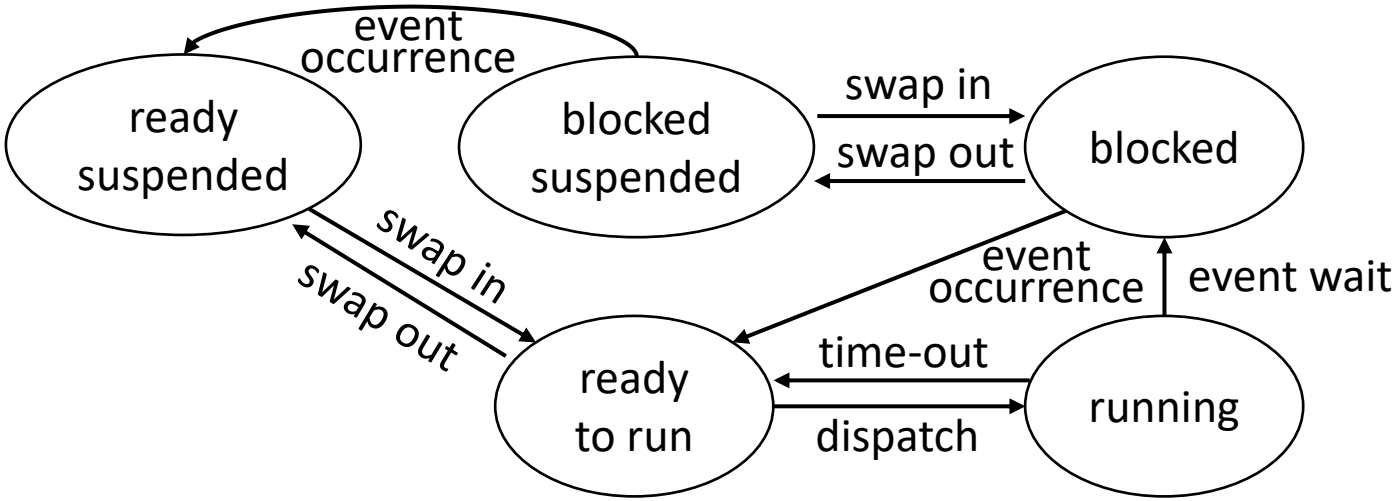
# Modello di esecuzione di un processo



# Modello di esecuzione di un processo

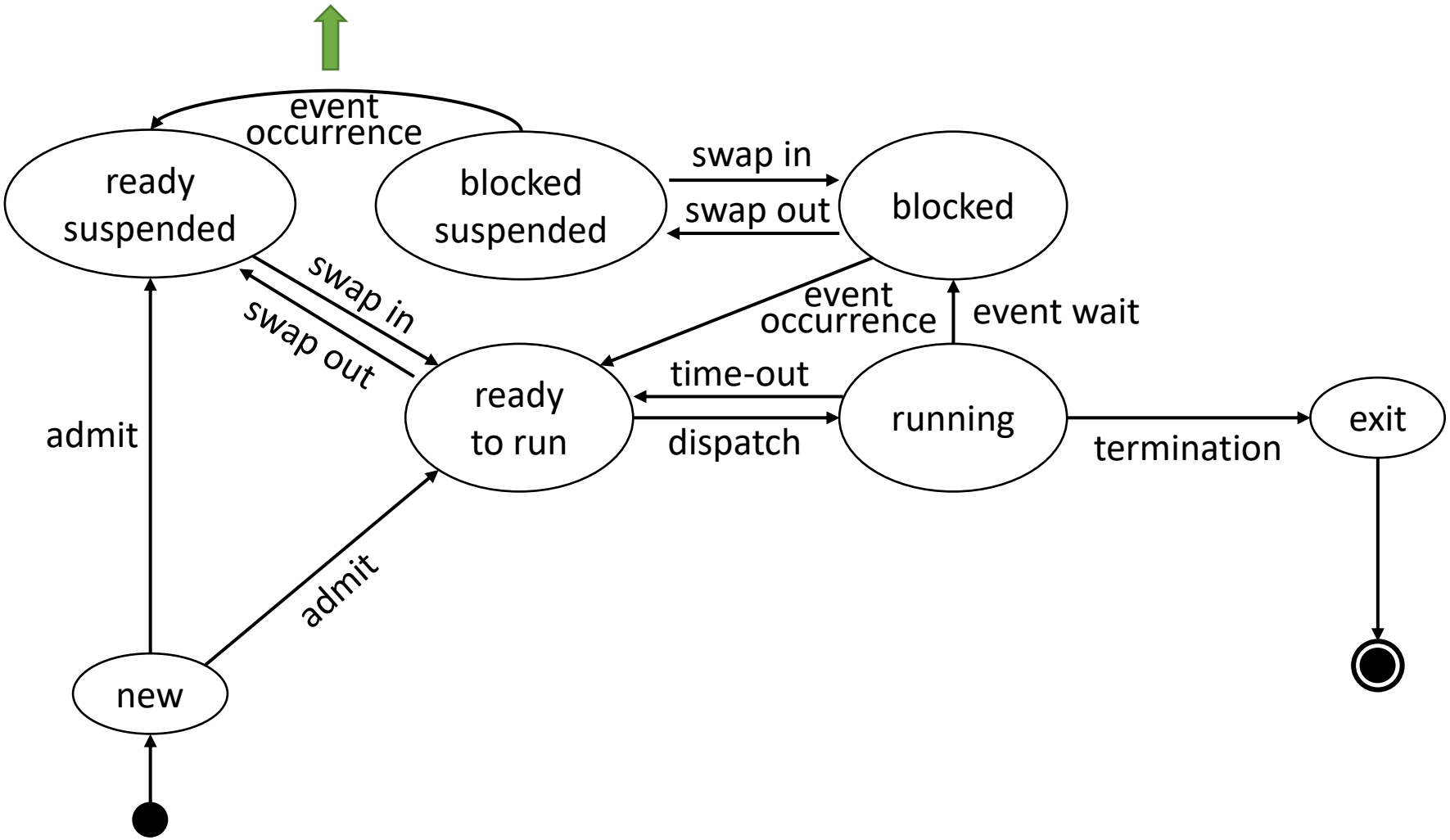


# Modello di esecuzione di un processo

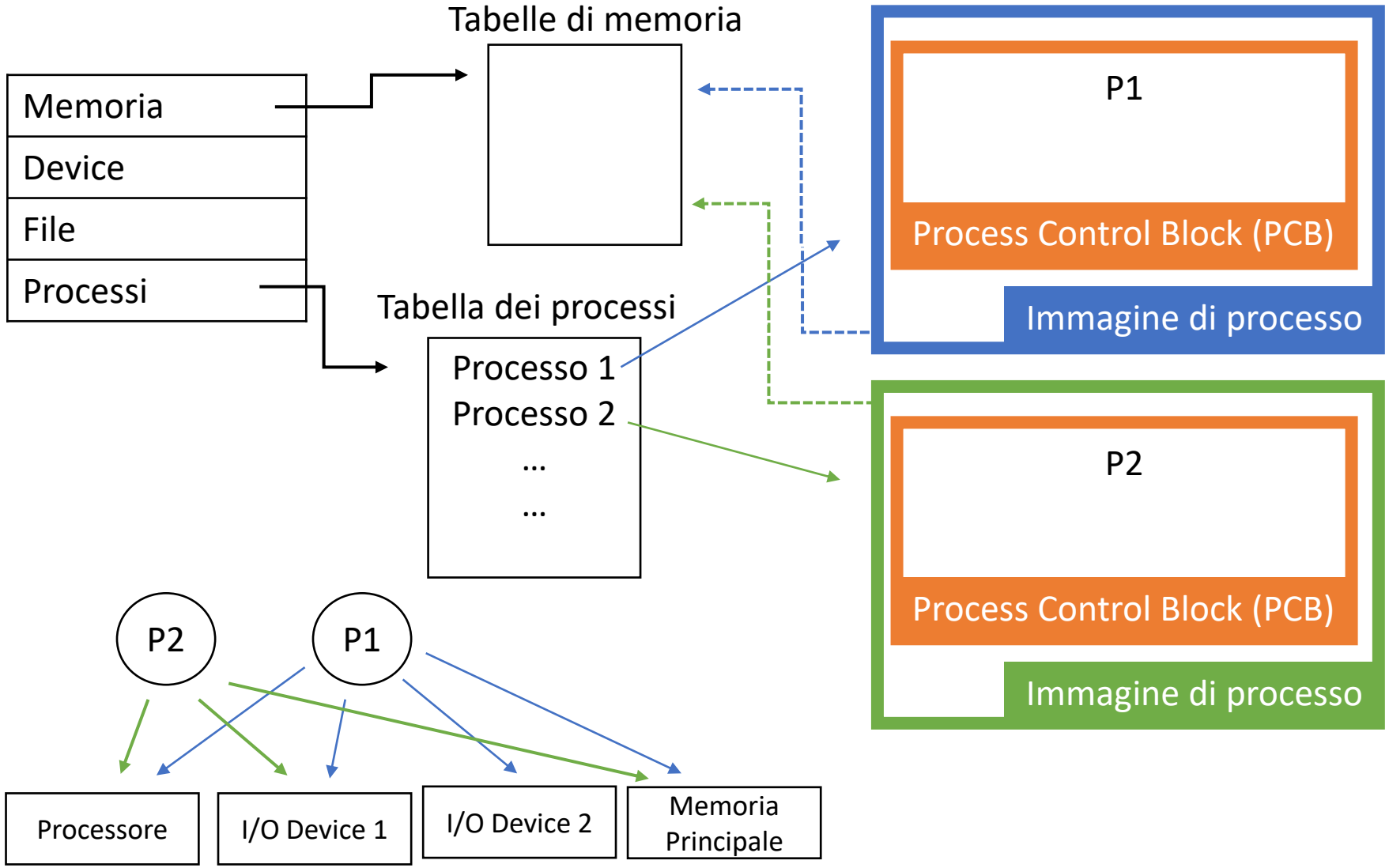


# Modello di esecuzione di un processo

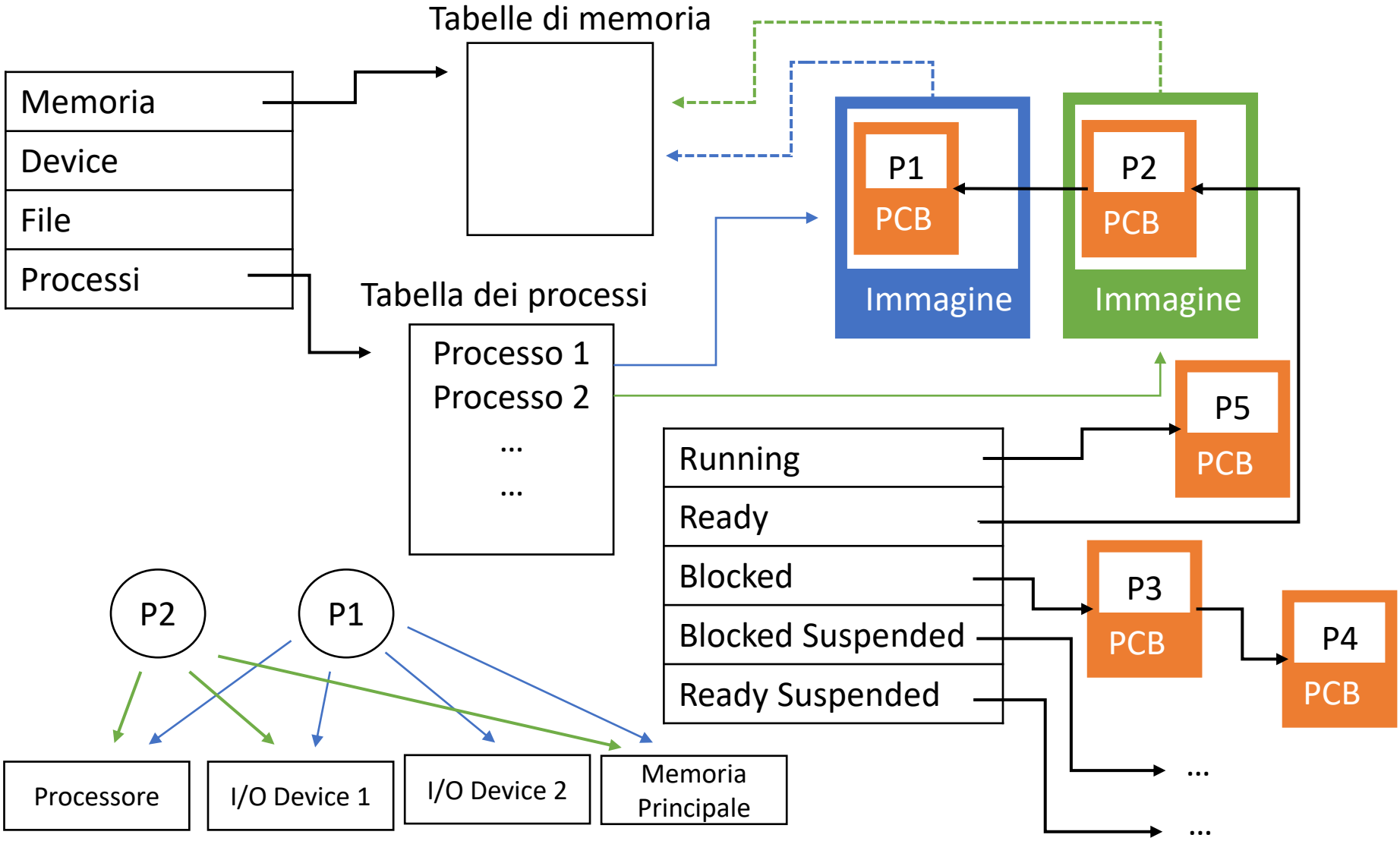
Il sistema operativo ha bisogno di metadati in memoria riguardo lo stato di un processo non in memoria



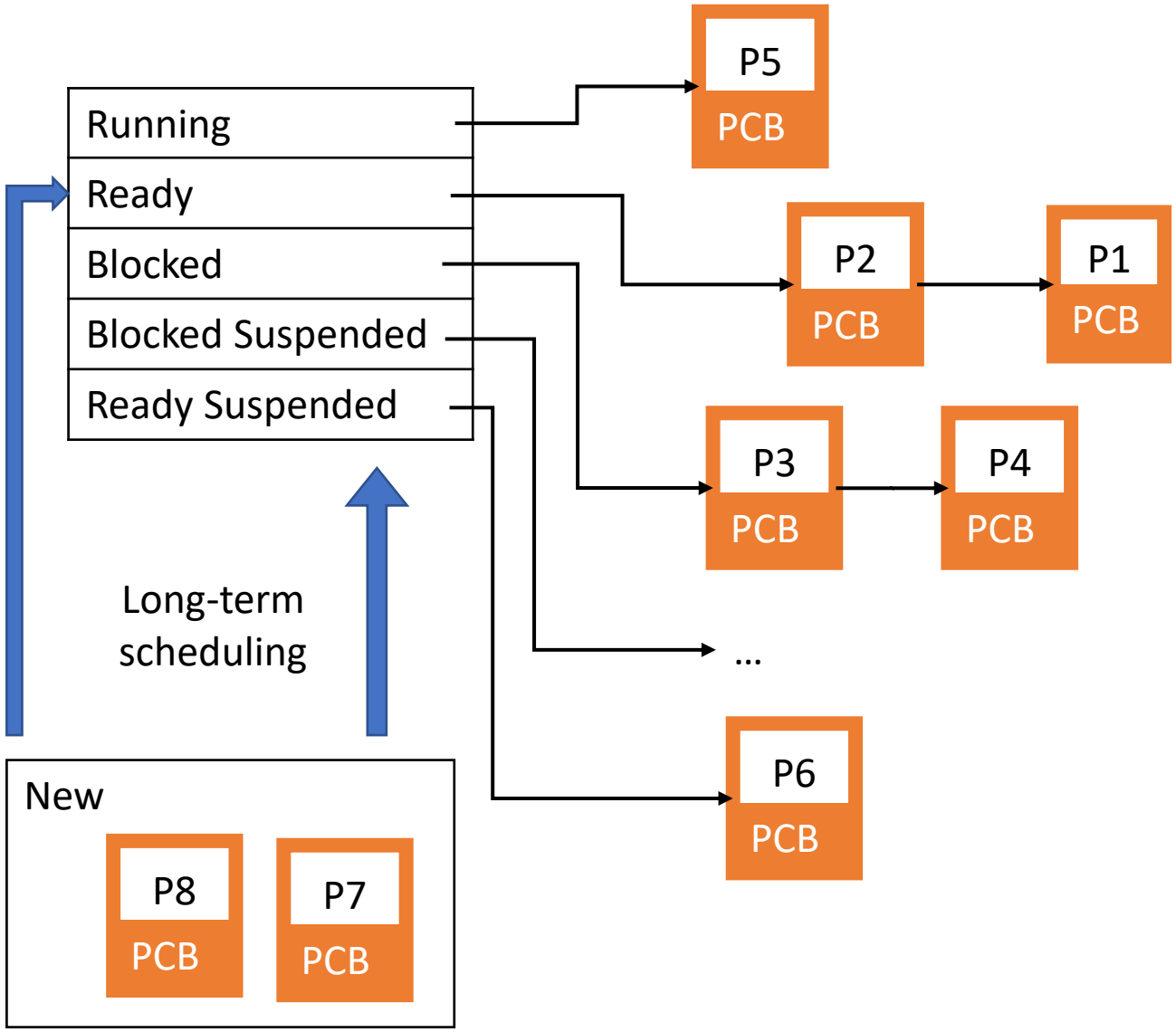
# Processi e strutture dati



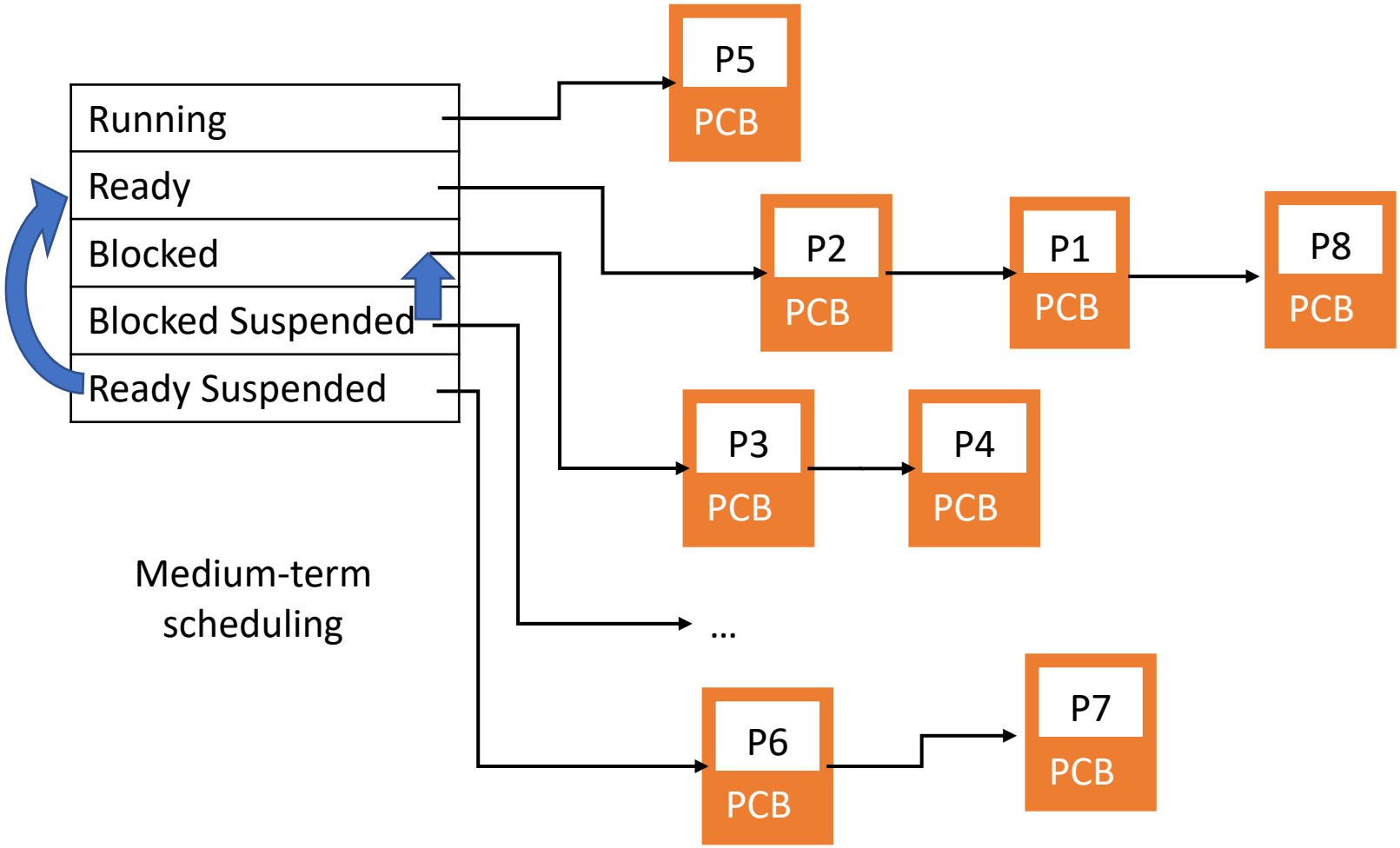
# Processi e strutture dati



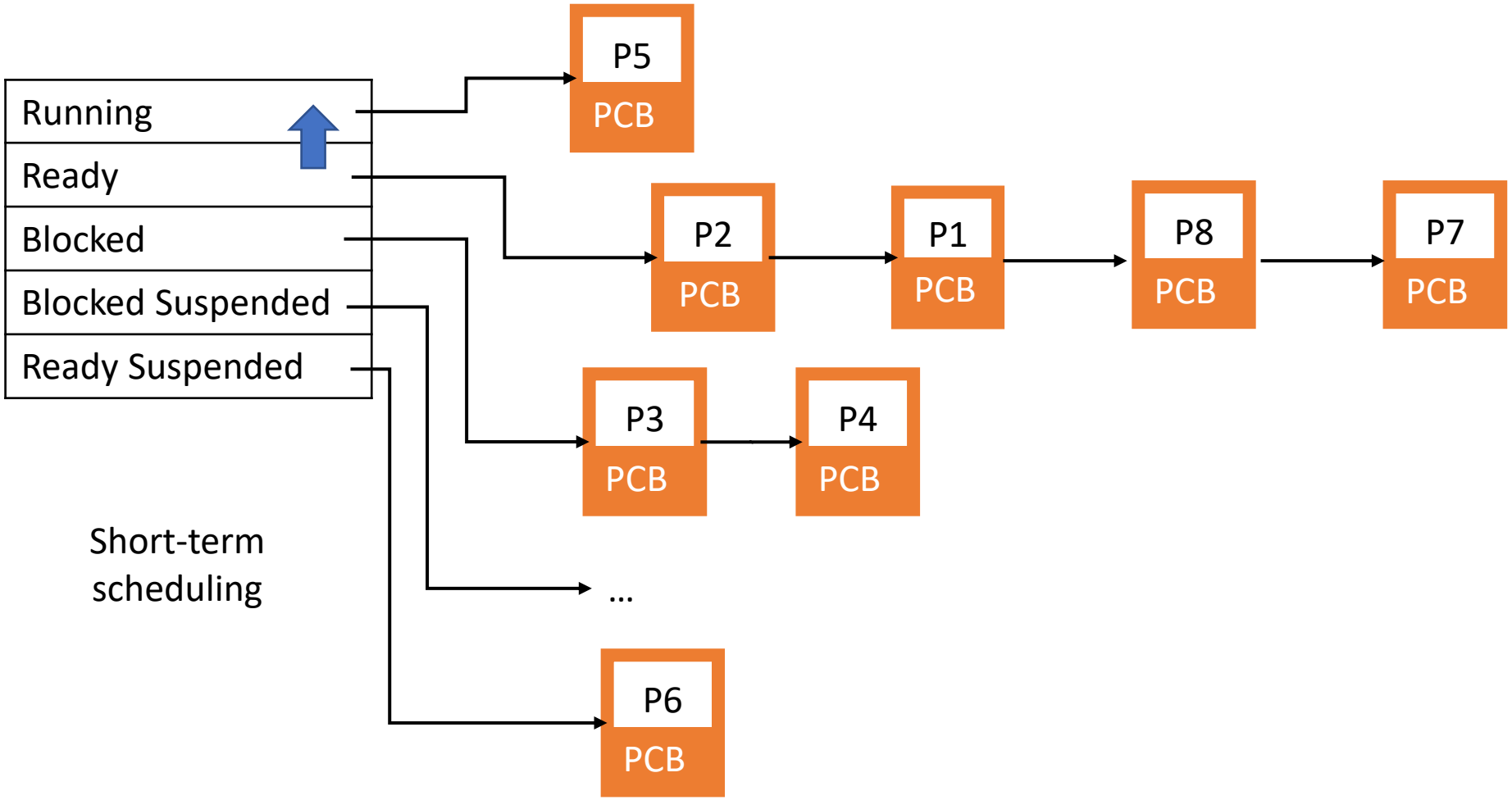
# Processi e strutture dati



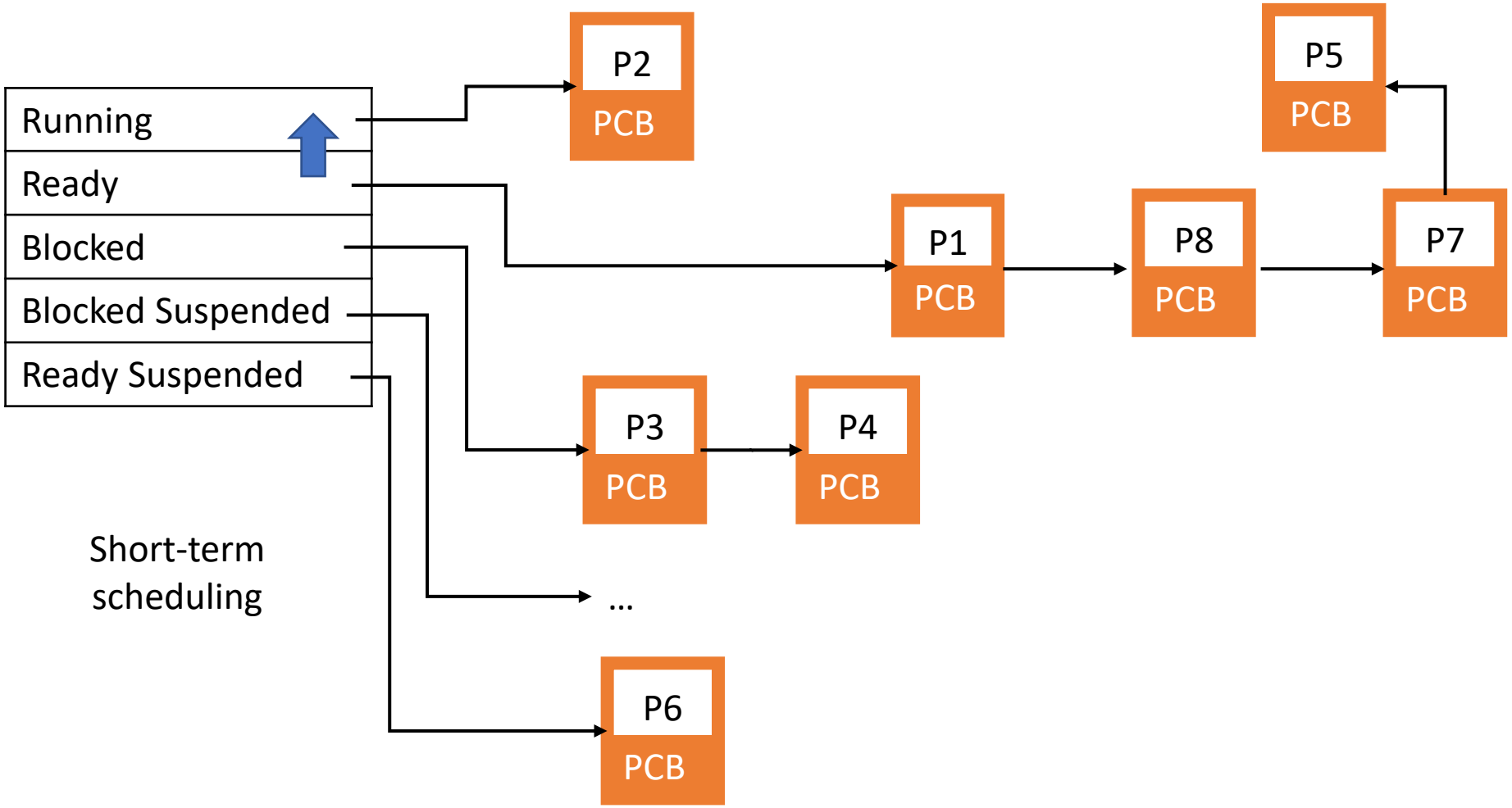
# Processi e strutture dati



# Processi e strutture dati

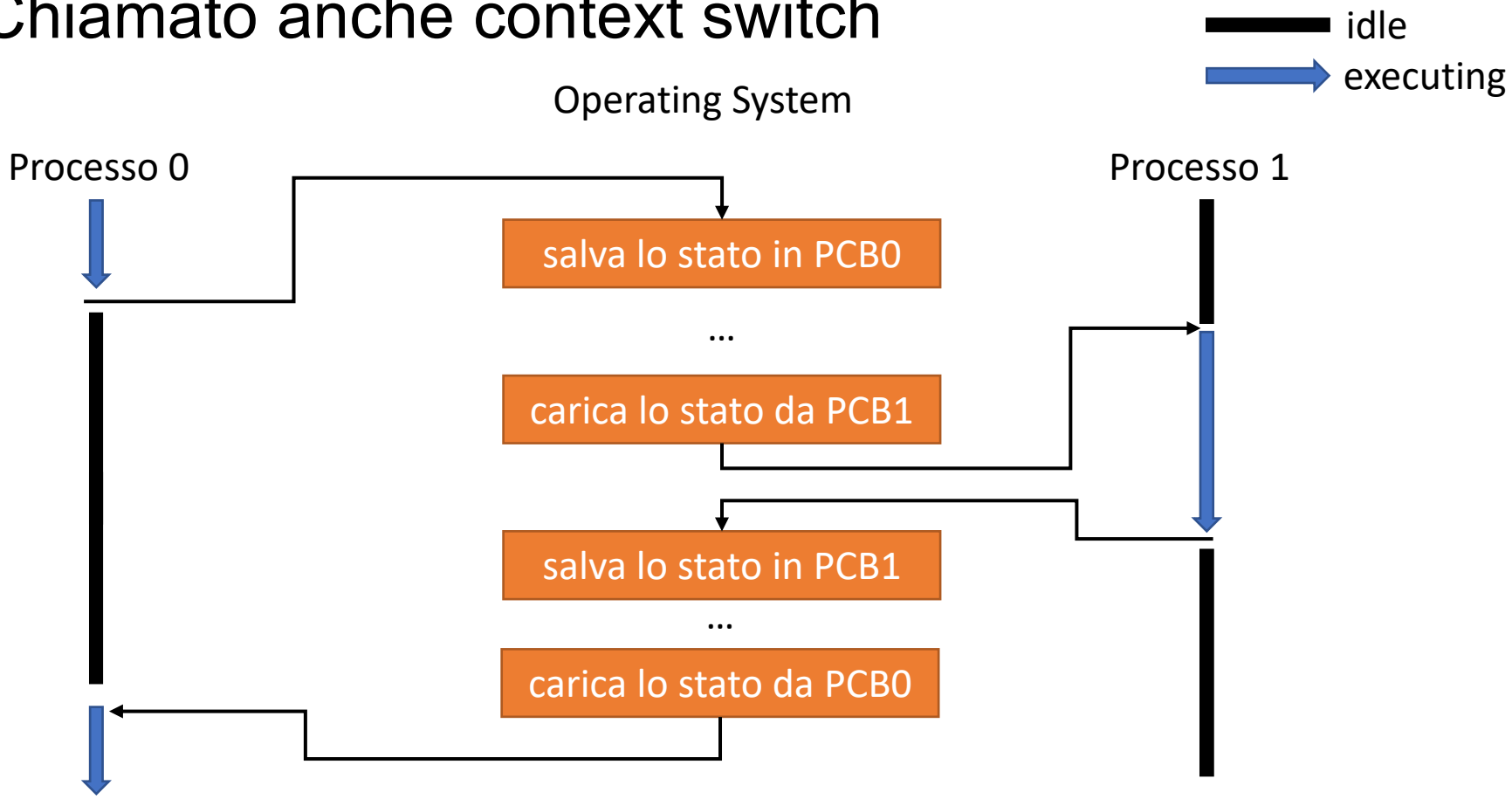


# Processi e strutture dati

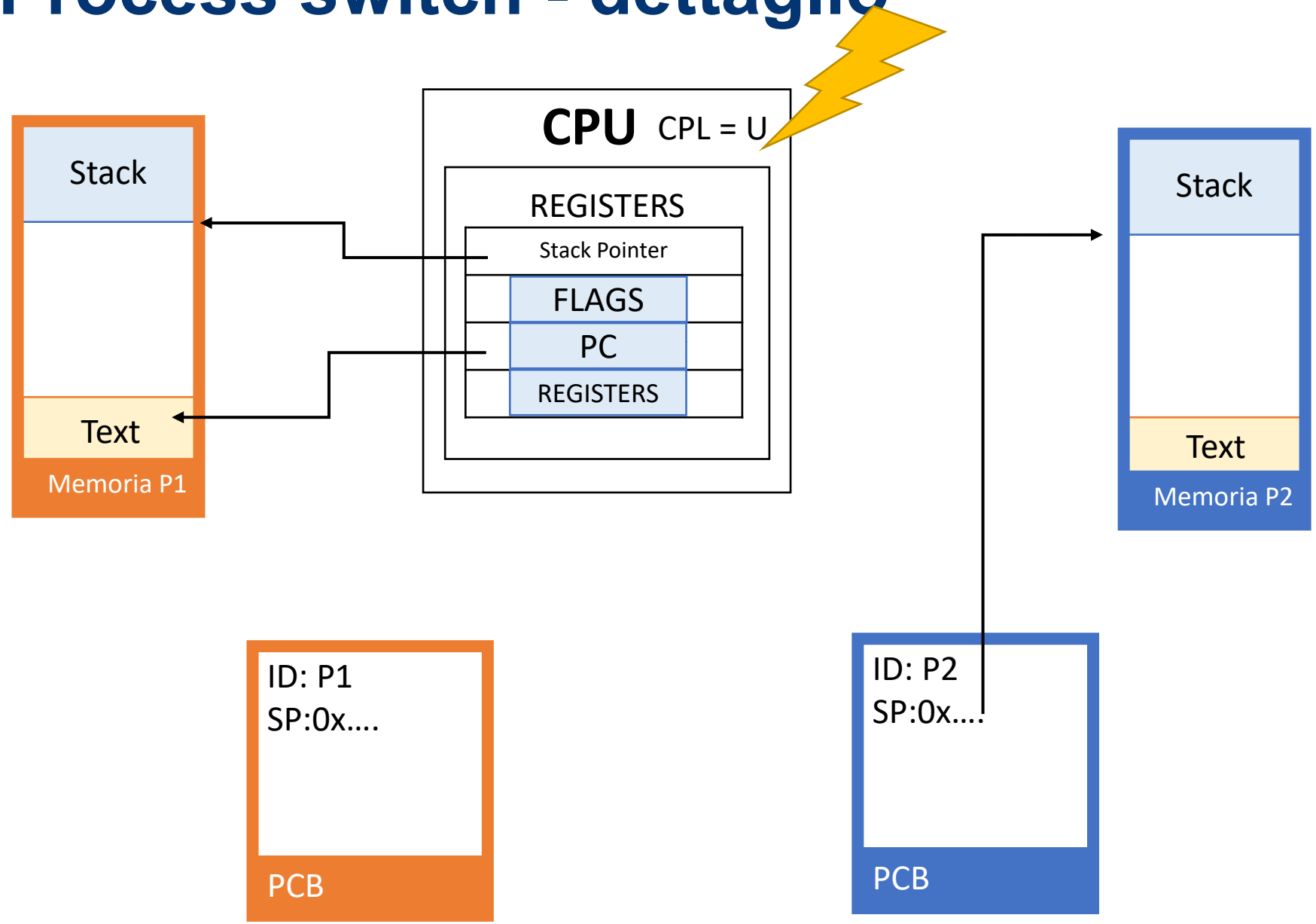


# Process switch

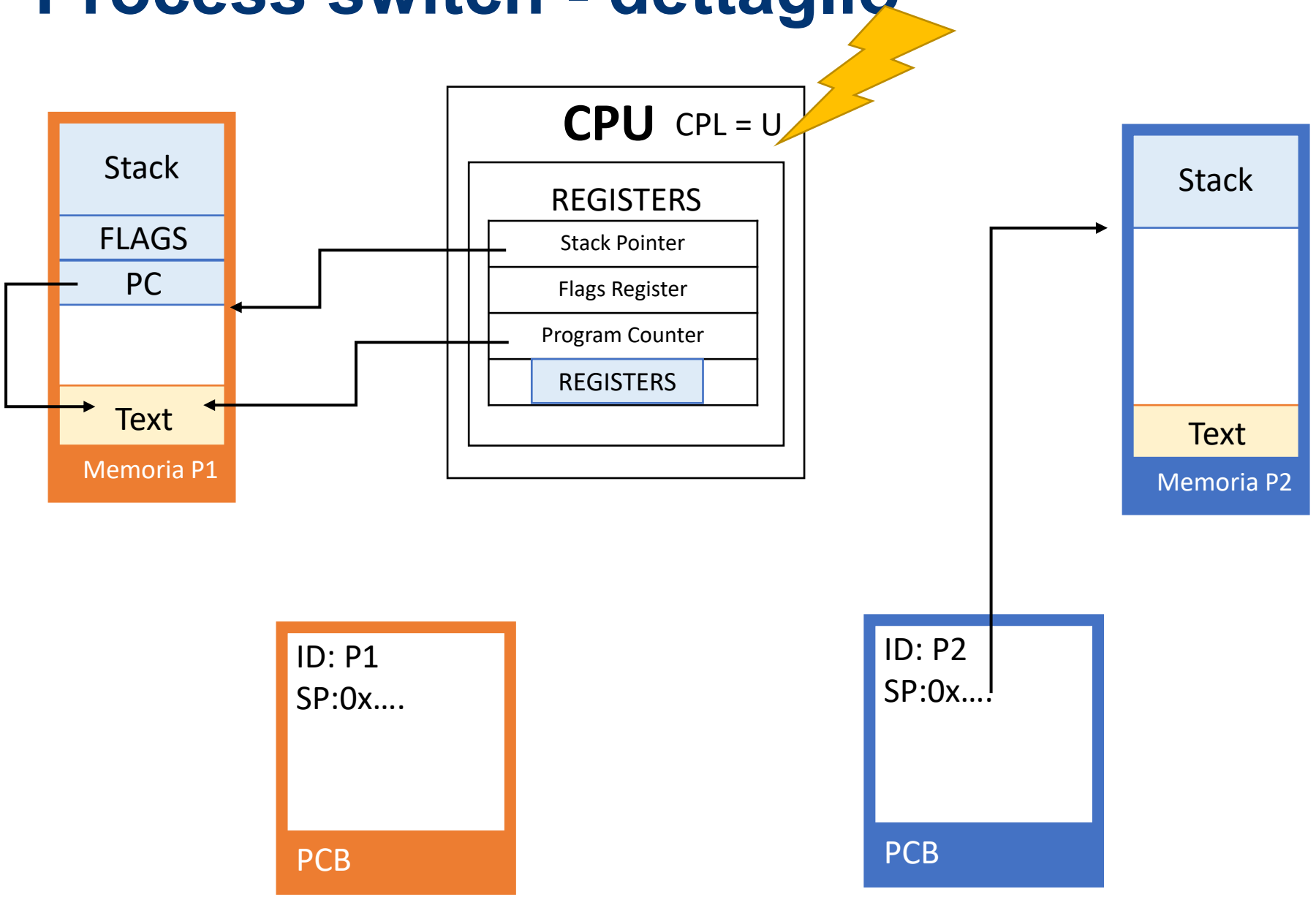
- Meccanismo per cambiare il processo in esecuzione sul processore => cambiare contesto
- Chiamato anche context switch



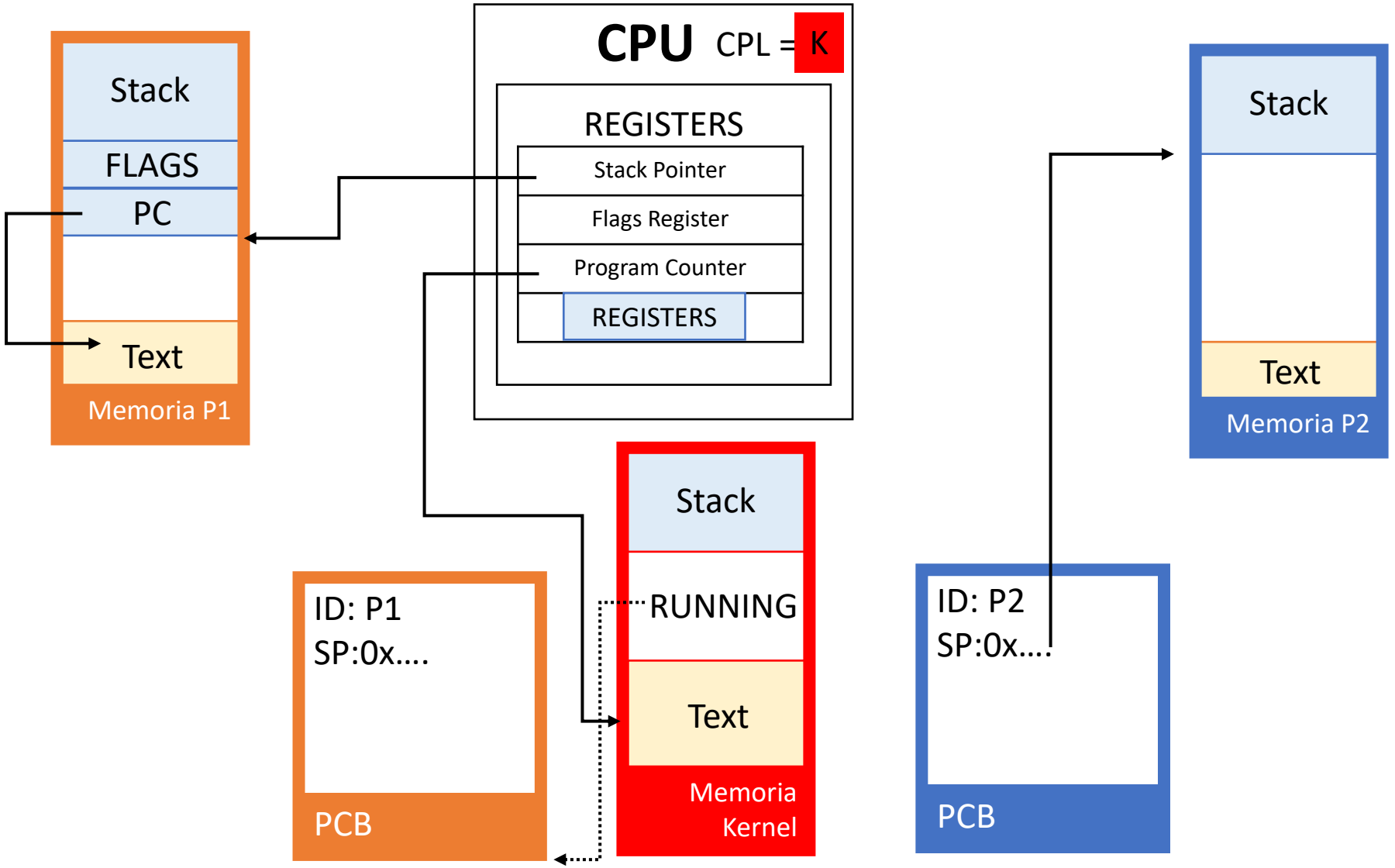
# Process switch - dettaglio



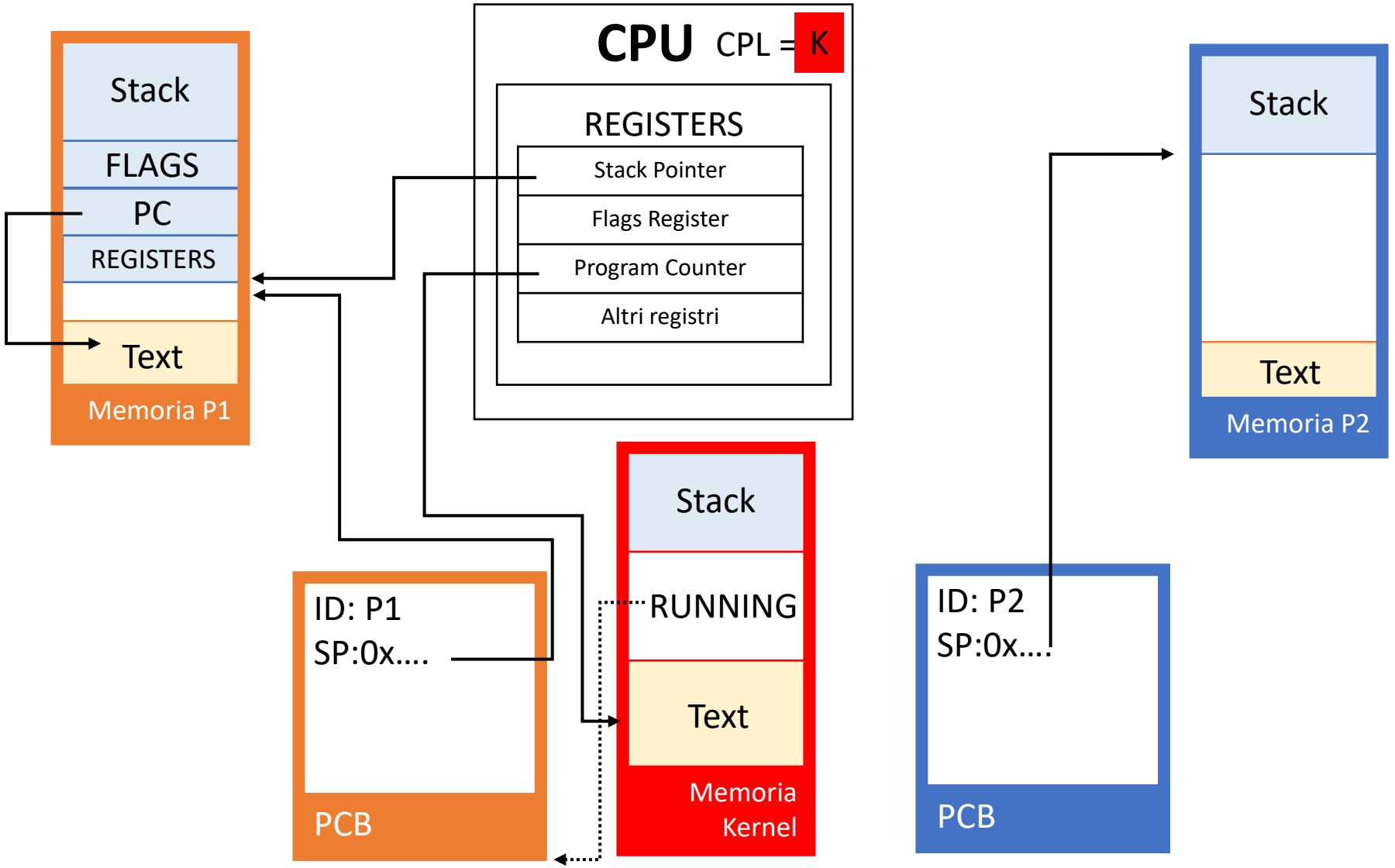
# Process switch - dettaglio



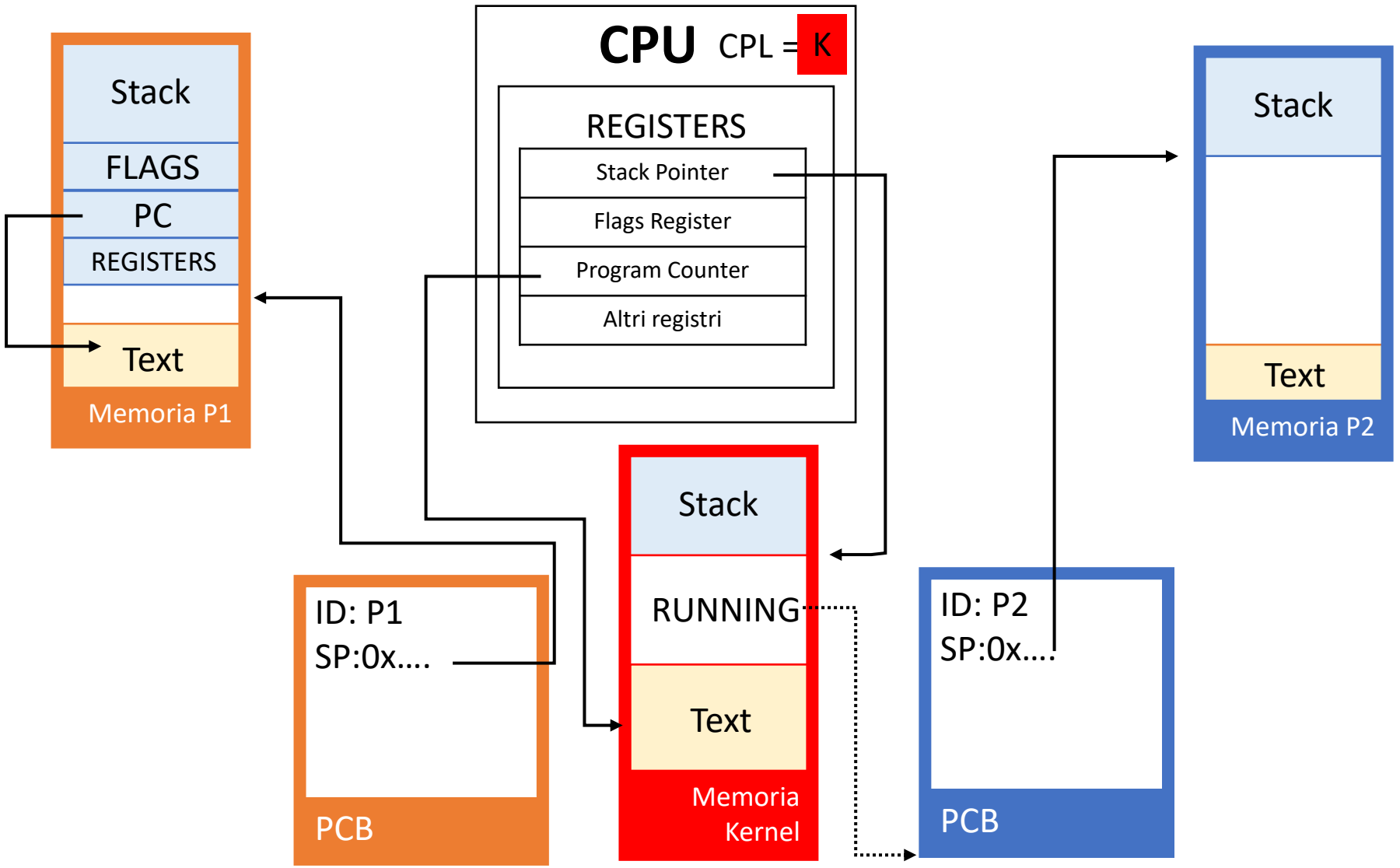
# Process switch - dettaglio



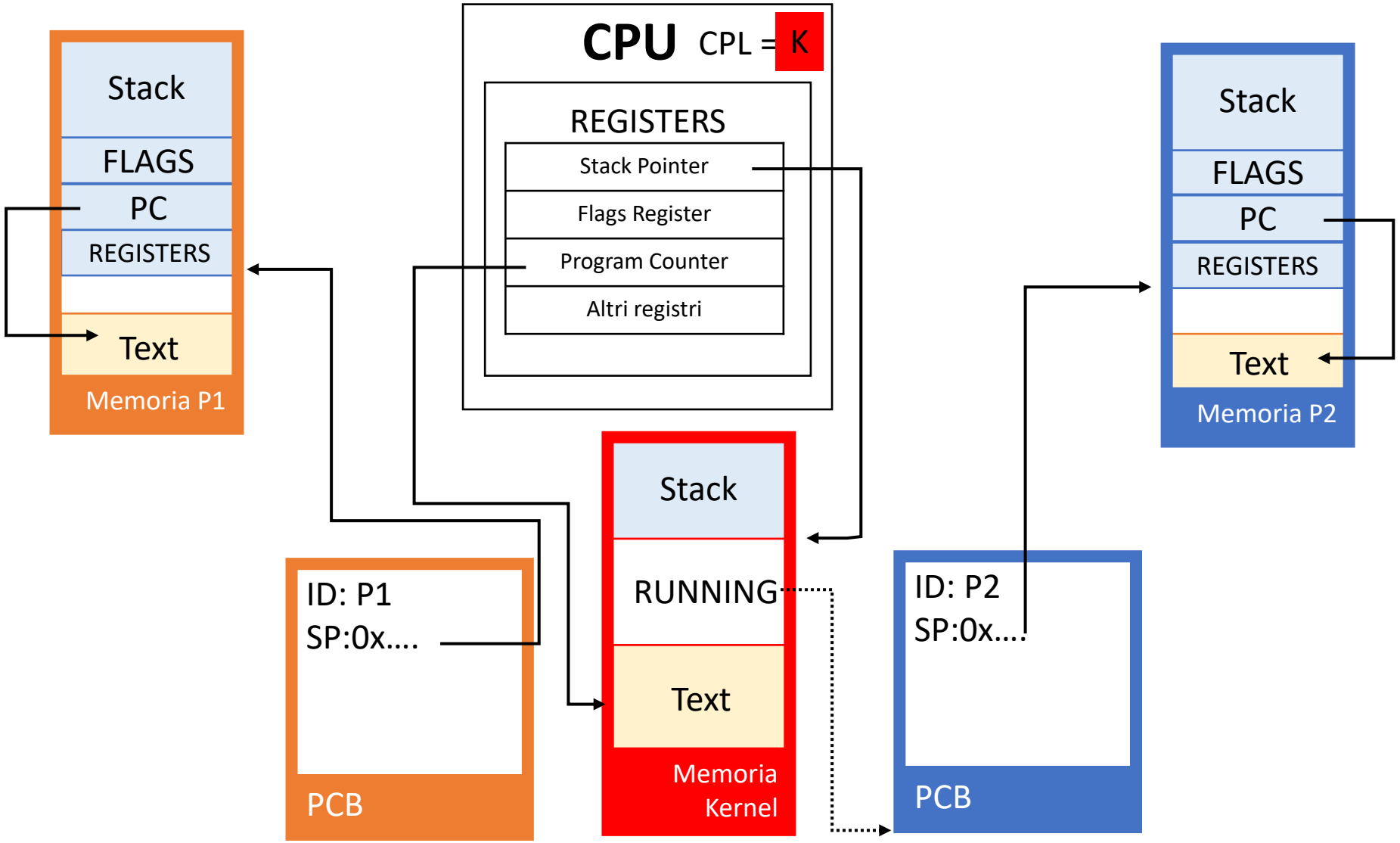
# Process switch - dettaglio



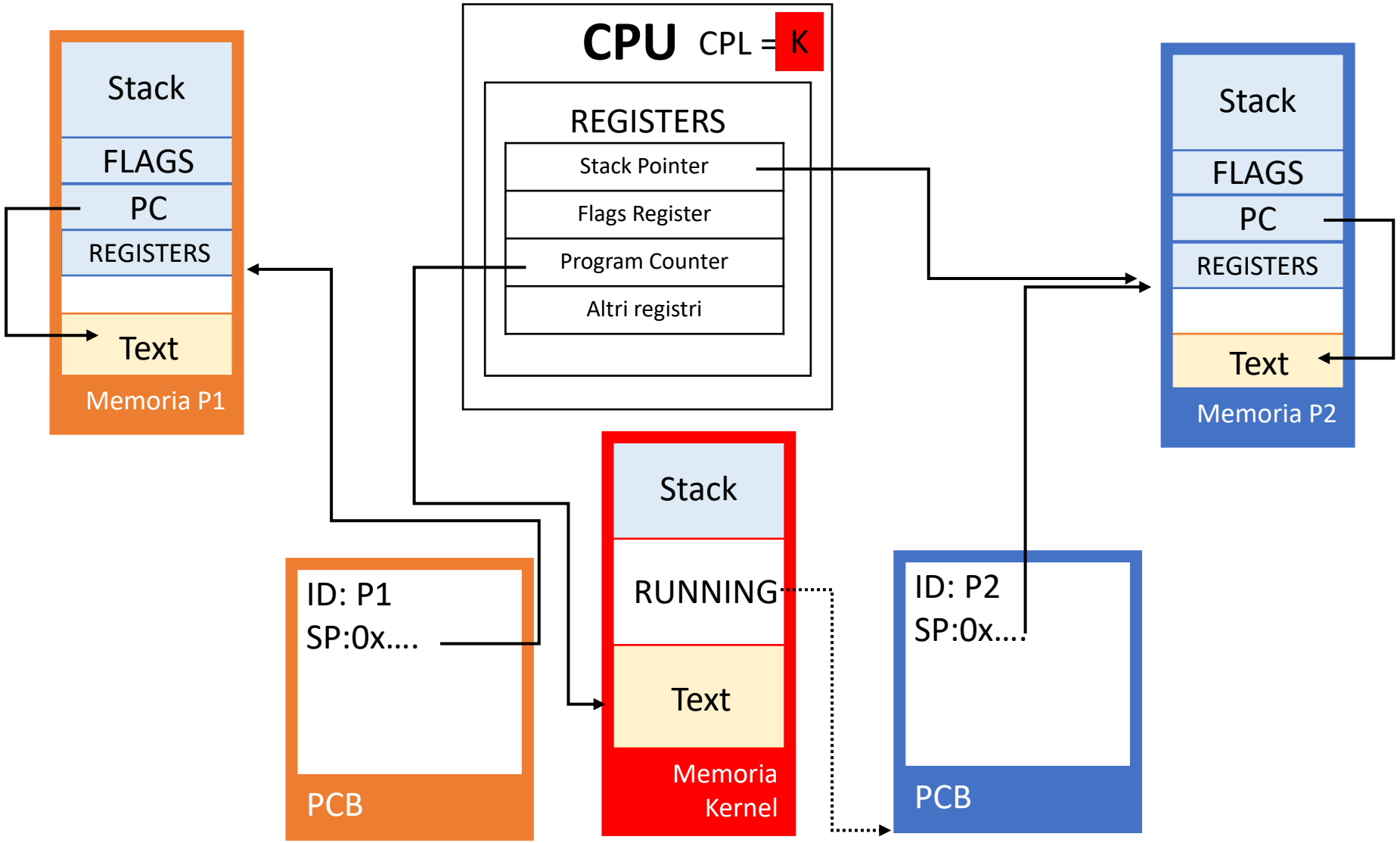
# Process switch - dettaglio



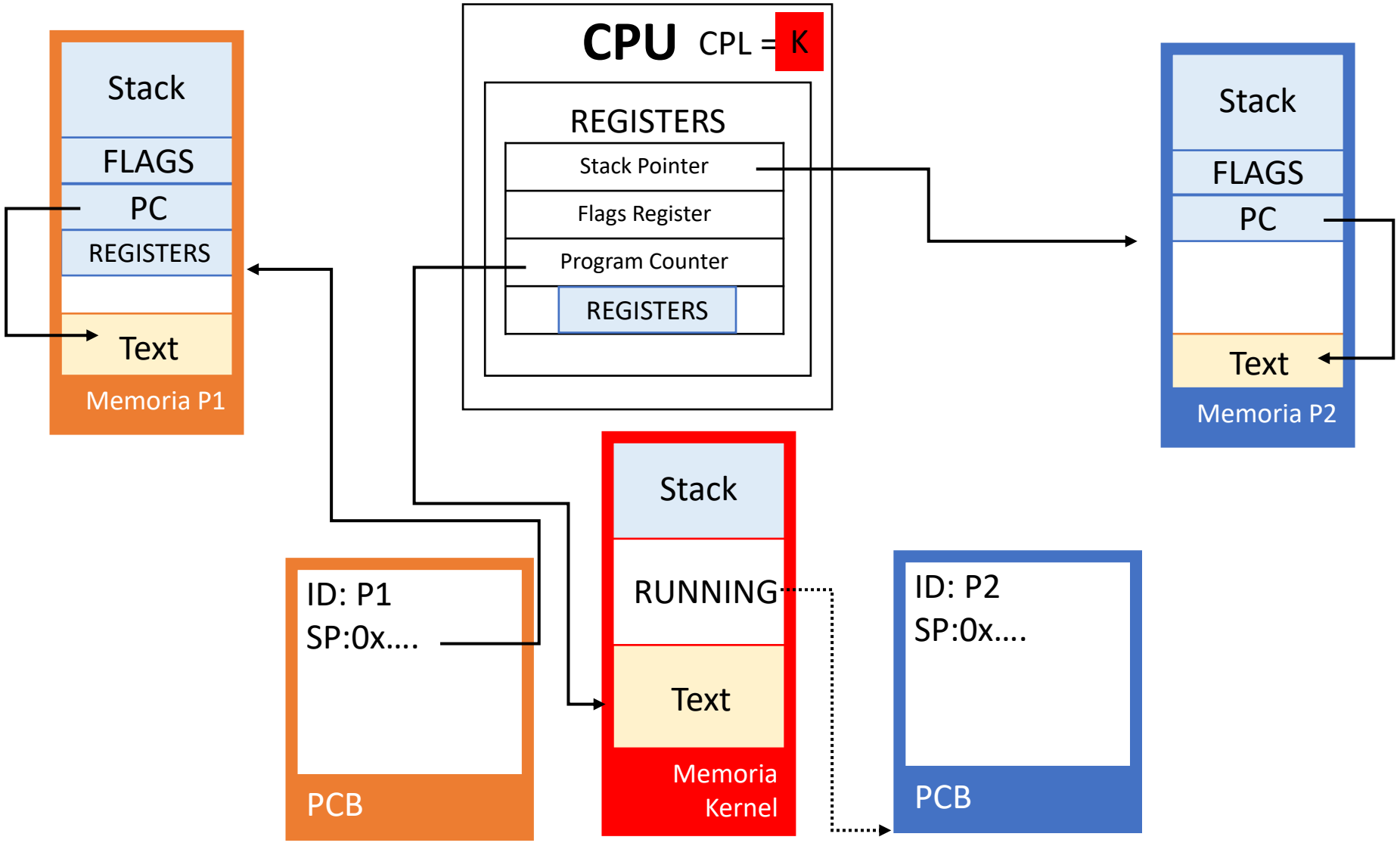
# Process switch - dettaglio



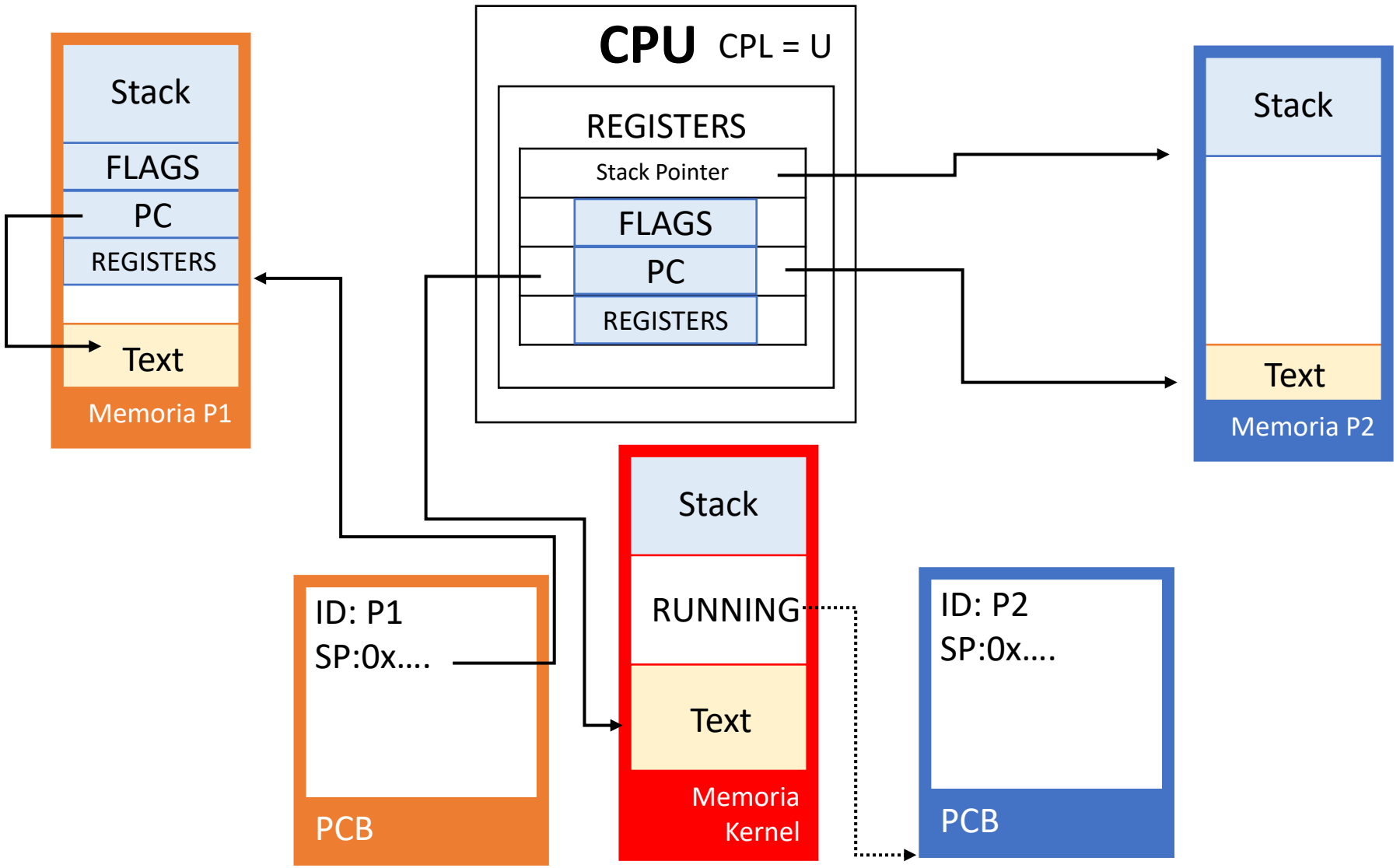
# Process switch - dettaglio



# Process switch - dettaglio



# Process switch - dettaglio



# Cambio di contesto vs Cambio di modo

## Cause diverse

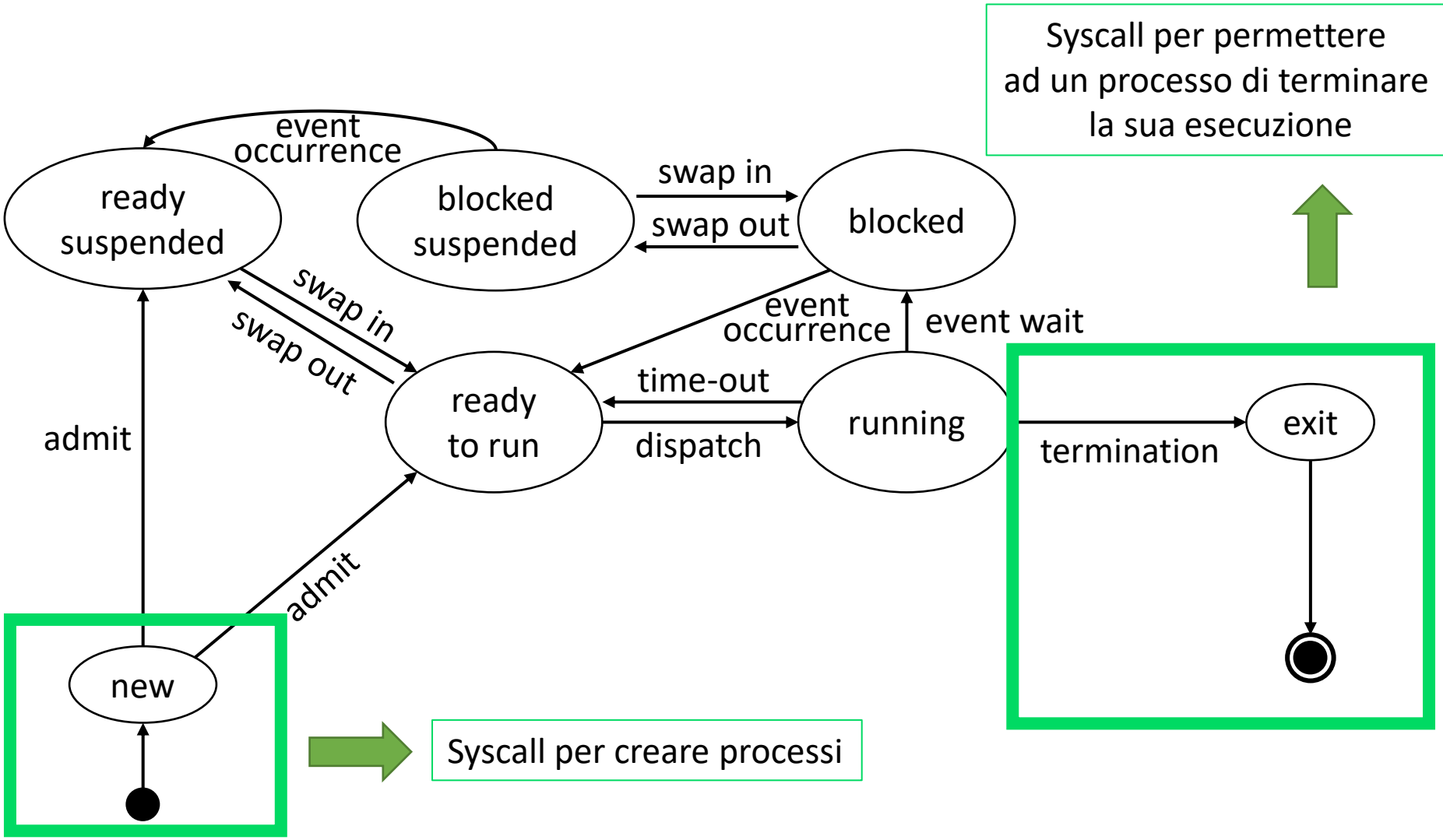
- Context switch
  - Interruzione da timer: viene attivato lo scheduler per cedere il controllo ad un altro processo
  - Interazione con I/O e conseguente attesa: viene attivato lo scheduler per cedere il controllo ad un altro processo
  - Errore non gestito: deattivazione del processo corrente : viene attivato lo scheduler per cedere il controllo ad un altro processo
- Mode switch
  - Invocazione di un system call
  - Gestione di una interruzione

## Esigenze diverse

- Context switch: necessità di salvare/ripristinare **tutto** il contesto
- Mode switch: necessità di salvare/rispristinare una **porzione** di contesto

## Cambio di modo **NON** implica Context/Process switch

# Modello di esecuzione di un processo



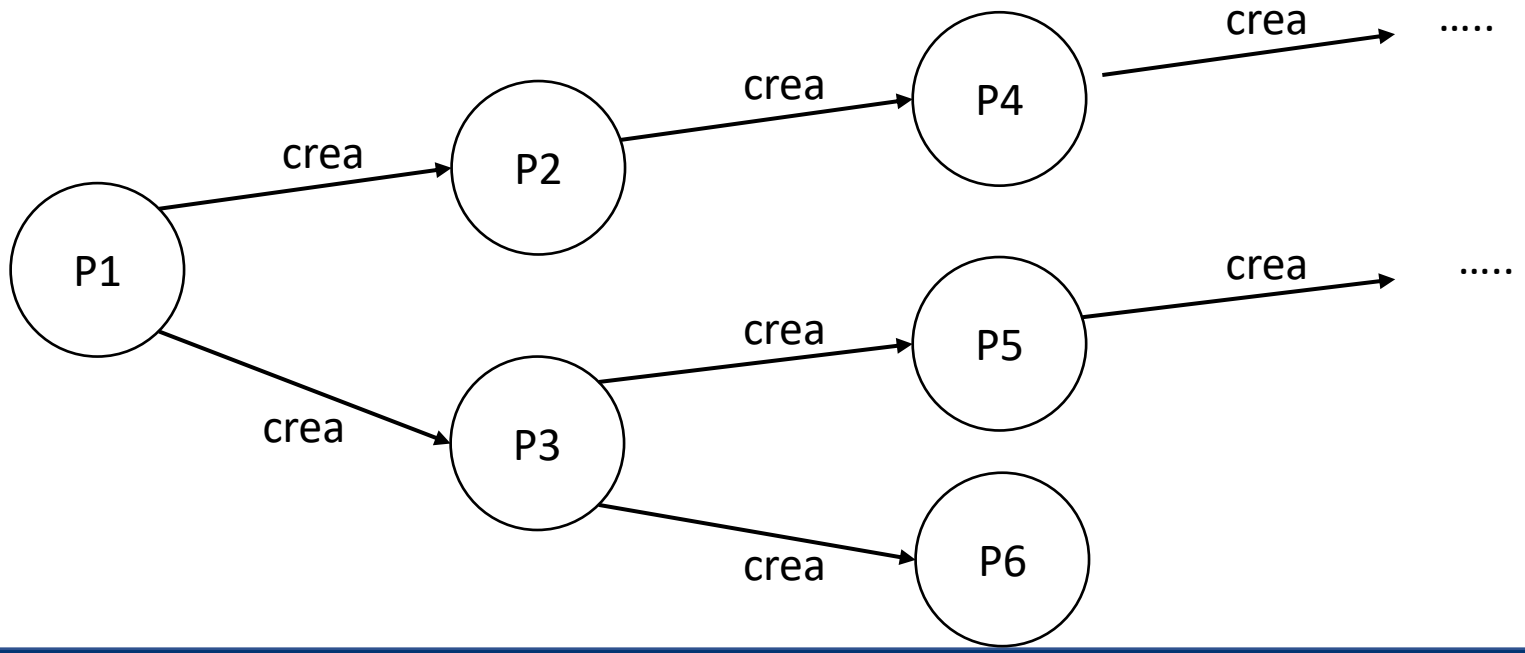
# (Alcuni) Servizi di sistema per gestire i processi

| |
|---|
| Creare un processo |
| Permettere ad un processo di terminare la propria esecuzione |

# Gerarchie di processi

Esiste una system call per creare un processo

- Un processo può creare uno o più processi
- A loro volta tali processi possono creare altri processi…



# Elementi caratterizzanti di un processo

- Un processo è associato a:
  - Un programma
  - I dati su cui opera
  - Almeno uno stack
  - Dati di contesto (contenuti nei registri di processore)
  - Le risorse hardware di cui ha richiesto l'accesso
  - Un identificativo **del processo, del processo genitore, dei processi figli**
  - Statistiche
  - Uno stato (e.g., running)

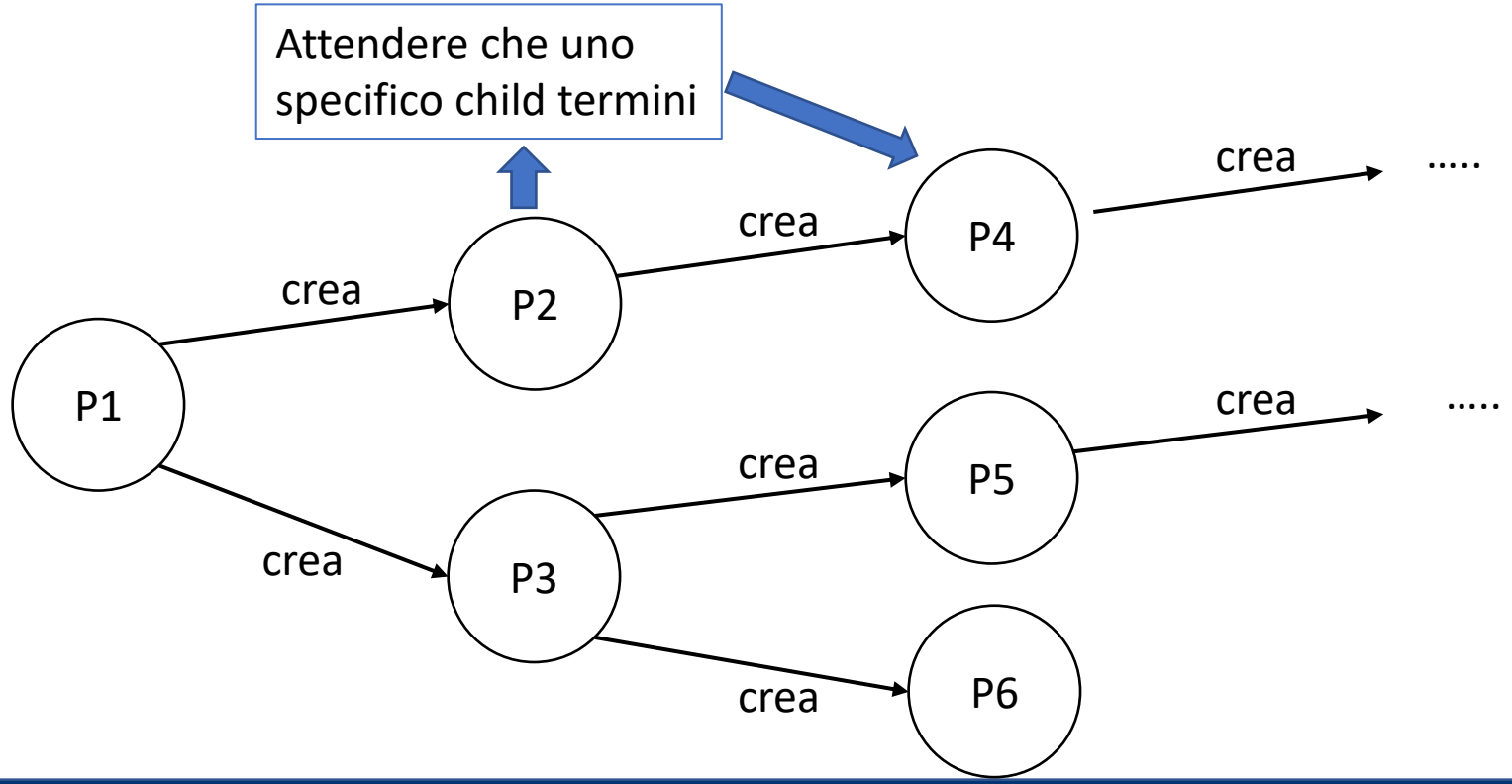
Process Control Block (PCB)

Immagine di processo

# Gerarchie di processi

Esiste una system call per creare un processo

- Un processo può creare uno o più processi
- A loro volta tali processi possono creare altri processi…

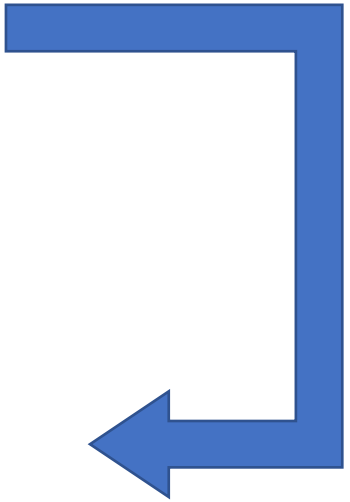


# (Alcuni) Servizi di sistema per gestire i processi

| DESCRIZIONE | UNIX/LINUX |
|---|---|
| Creare un processo | fork |
| Permettere ad un processo di terminare la propria esecuzione | exit |
| Attendere la terminazione di processo figlio | wait |

# (Alcuni) Servizi di sistema per gestire i processi

| DESCRIZIONE | UNIX/LINUX |
|---|---|
| Creare un processo | fork |
| Permettere ad un processo di terminare la propria esecuzione | exit |
| Attendere la terminazione di processo figlio | wait |



Standard POSIX:
https://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html

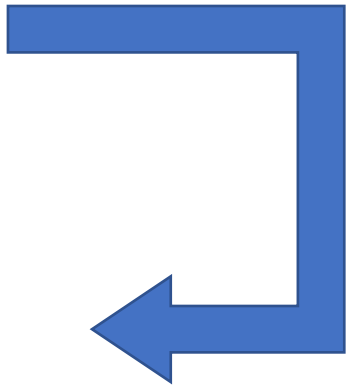
**NAME**
fork - create a new process

**SYNOPSIS**
```
#include <unistd.h>
pid_t fork(void);
```

**DESCRIPTION**
*The fork() function shall create a new process.*
*The new process (child process) shall be an **exact copy** of the calling process (parent process) except as detailed below:*
- *The child process shall have a unique process ID.*
- *…*

# (Alcuni) Servizi di sistema per gestire i processi

| DESCRIZIONE | UNIX/LINUX |
|---|---|
| Creare un processo | fork |
| Permettere ad un processo di terminare la propria esecuzione | exit |
| Attendere la terminazione di processo figlio | wait |



Standard POSIX:
https://pubs.opengroup.org/onlinepubs/9699919799/functions/exit.html

**NAME**
fork - create a new process

**SYNOPSIS**
```
#include <stdlib.h>
void exit(int status);
```

**DESCRIPTION**
*The value of status may be 0, EXIT_SUCCESS, EXIT_FAILURE,or any other value, though only the least significant 8 bits (that is, status & 0377) shall be available from wait().*

*……..*
*Finally, the process shall be terminated ….*

# (Alcuni) Servizi di sistema per gestire i processi

| DESCRIZIONE | UNIX/LINUX |
|---|---|
| Creare un processo | fork |
| Permettere ad un processo di terminare la propria esecuzione | exit |
| Attendere la terminazione di processo figlio | wait |

Standard POSIX:
https://pubs.opengroup.org/onlinepubs/9699919799/functions/wait.html

**NAME**
fork - create a new process

**SYNOPSIS**
```
#include <sys/wait.h>
pid_t
wait(int *status_location);
```

**DESCRIPTION**
*The wait() … functions shall obtain status information … pertaining to one of the caller's child processes. The wait() function obtains status information for process termination from any child process.*

*……*

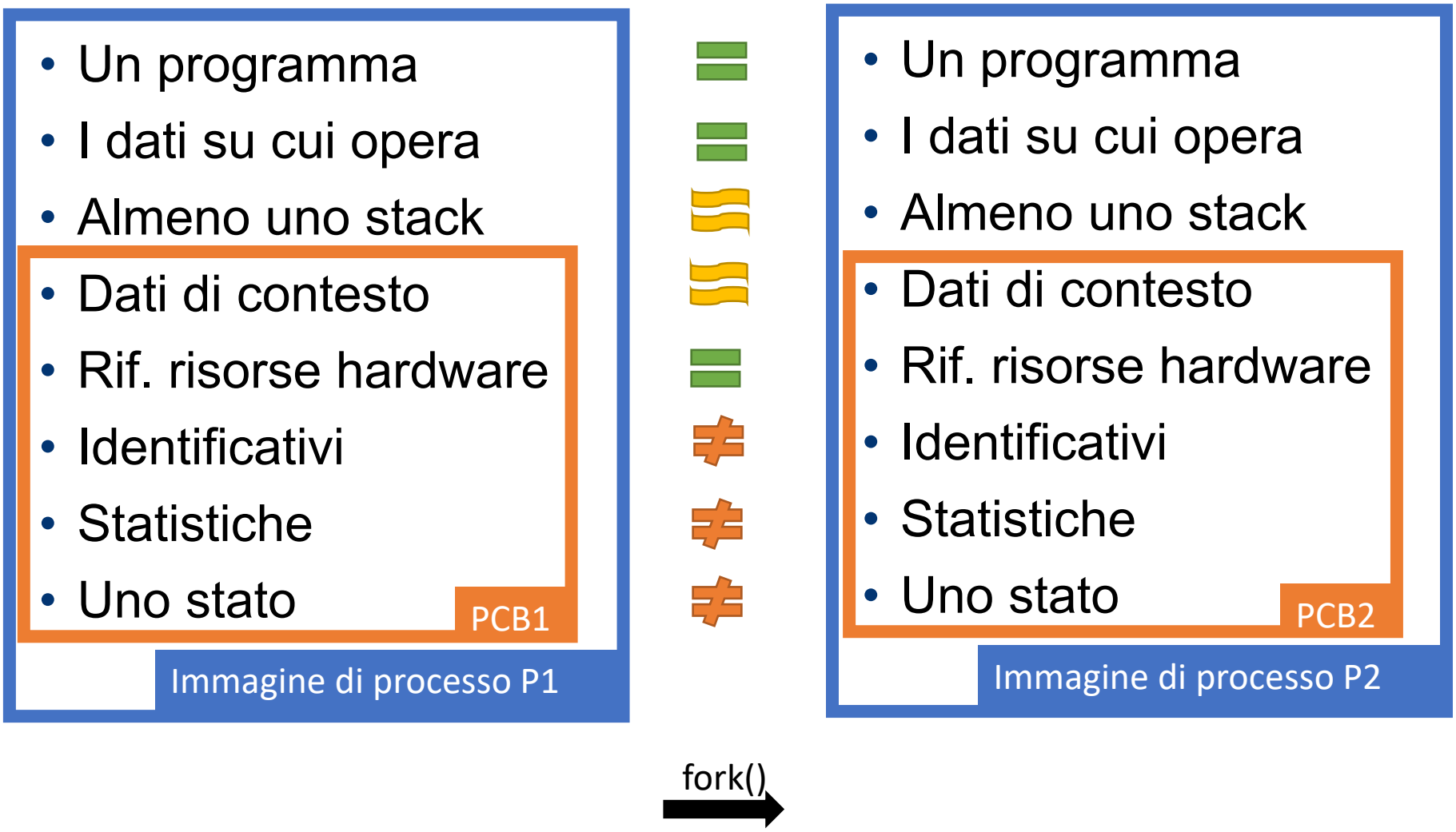
*The wait() function shall cause the calling thread to become blocked until status information generated by child process termination is made available….*

# (Alcuni) Servizi di sistema per gestire i processi

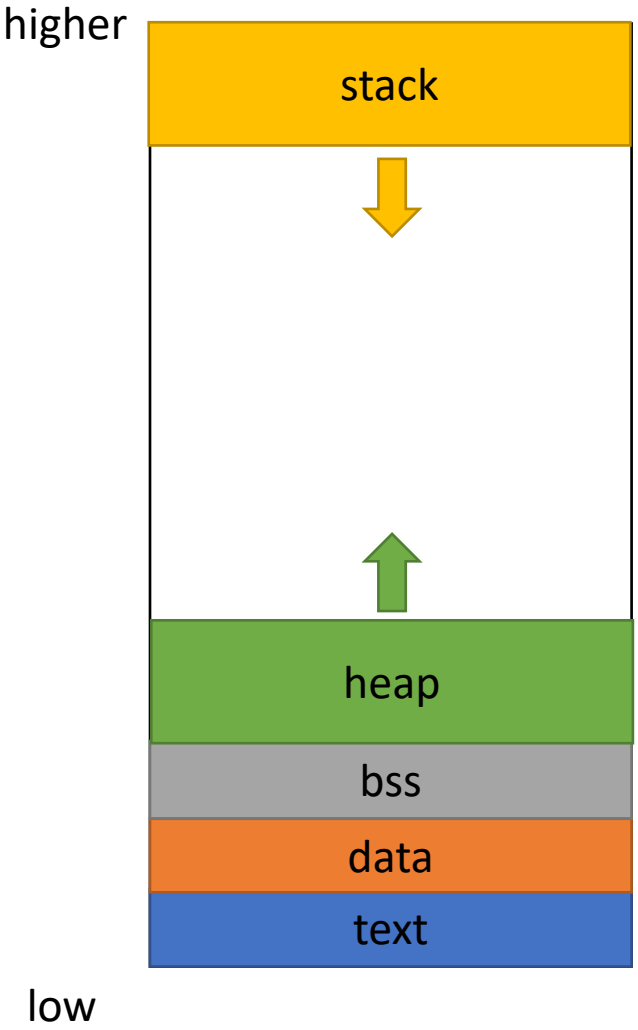
```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

void main(){
  int res, status;
  printf("I'm a process and I'm going to create a child\n");
  res = fork();
  if(res < 0) printf("I cannot create a child");
  else if(res == 0){
    printf("I'm the child!\n");
    exit(0);
  }
  else{
    printf("I'm now a parent and I'll wait for my child to die...\n");
    wait(&status);
    printf("My child has invoked exit? %d\n", WIFEXITED(status));
    printf("My child has invoked exit(%d)\n", WEXITSTATUS(status));
  }
  printf("My child is dead, so it's my time to die\n");
  exit(0);
}
```

# Fork

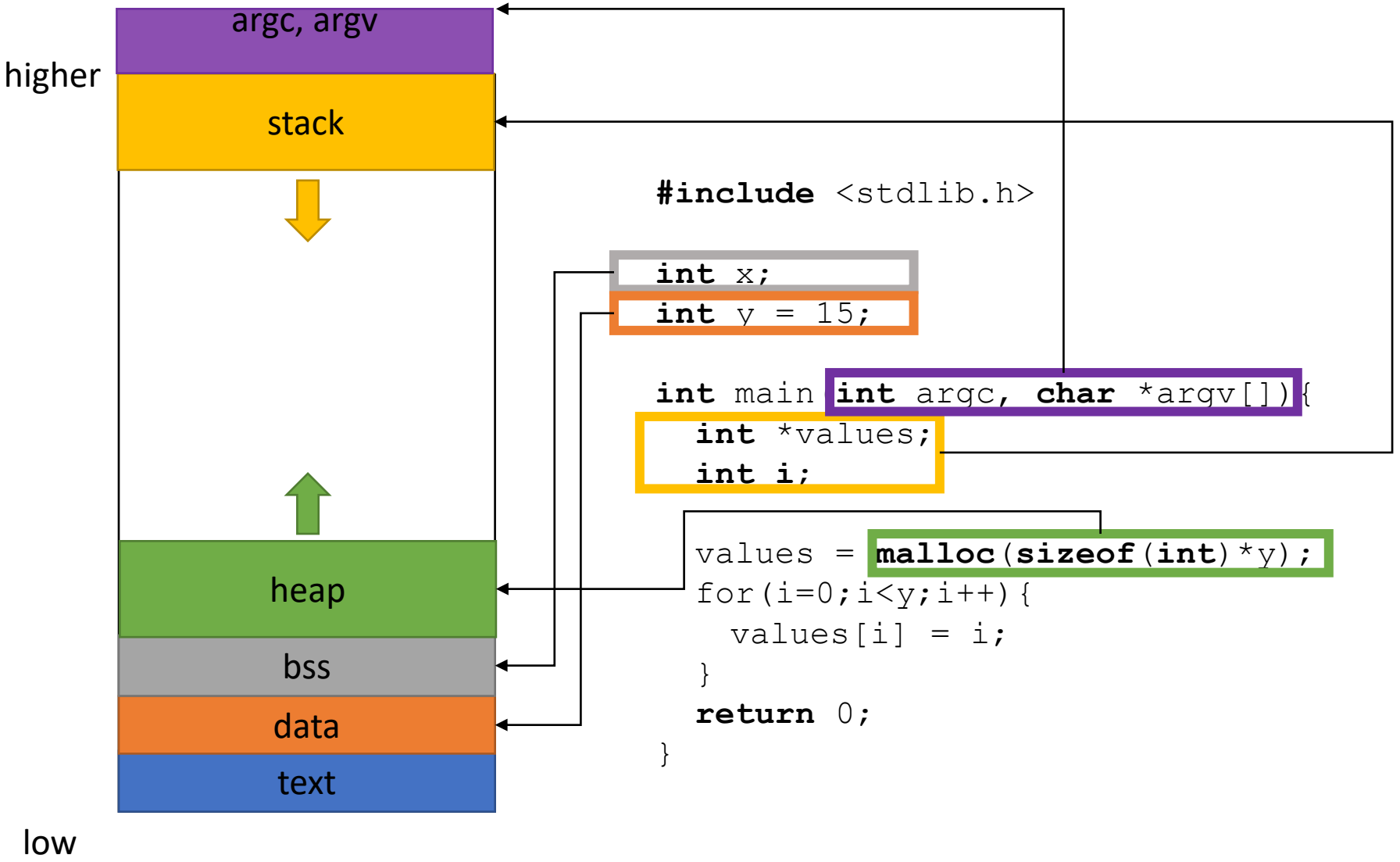

• Un programma
• I dati su cui opera
• Almeno uno stack
• Dati di contesto
• Rif. risorse hardware
• Identificativi
• Statistiche
• Uno stato
PCB1
Immagine di processo P1
• Un programma
• I dati su cui opera
• Almeno uno stack
• Dati di contesto
• Rif. risorse hardware
• Identificativi
• Statistiche
• Uno stato
PCB2
Immagine di processo P2
fork()

# Layout di un programma C




- **Text:** istruzioni eseguibili
- **Data:** dati inizializzati
- **Block started by symbol (BSS):** dati non inizializzati o initizializzati al valore zero
- **Heap:** sezione di dati allocati dinamicamente
- **Stack:** per chiamate a procedura, passaggio parametri, indirizzo di ritorno, variabili locali

# Layout di un programma C



# Sostituzione di programma

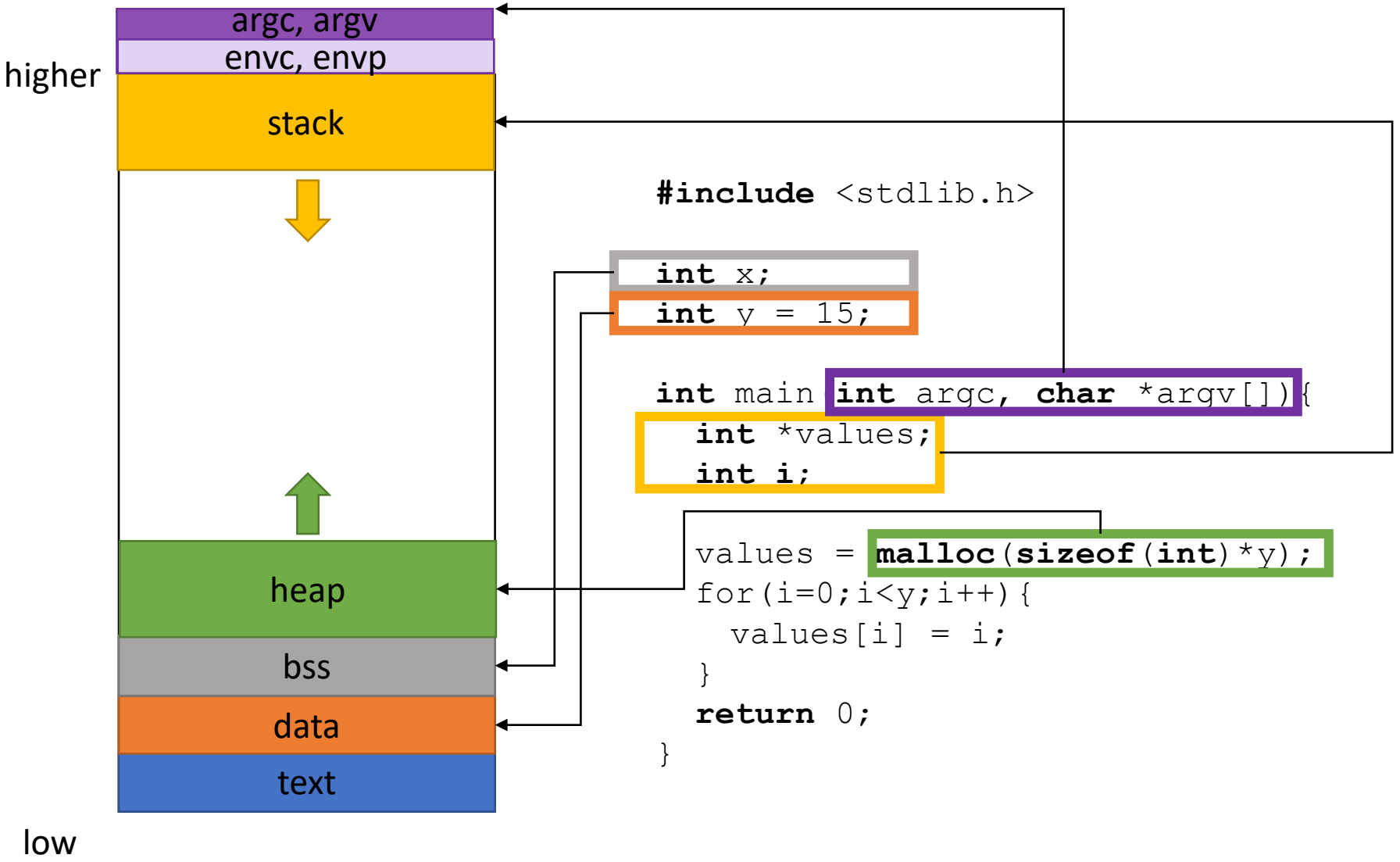
- Meccanismo per sostituire il programma associato al corrente processo di esecuzione
- Famiglia di funzioni **exec** permettono di definire:
  - il programma che sostituirà il codice del processo corrente
  - dove cercare il programma corrente (p)
  - i parametri da passare al programma come parametri multipli (l) o come array (v)
  - l'ambiente del nuovo processo (e)

```
SYNOPSIS
#include <unistd.h>
int execl(const char *pathname, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *pathname, const char *arg, ..., char *const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

# Ambiente

- Environment list: un array di stringhe (environment variables) nella forma **nome=valore**
- Permettono di configurare il comportamento di programma in relazione ad altro software (l'ambiente)
  - Dove cercare altri eseguibili/librerie
- Ad esempio in sistemi UNIX, al lancio di un eseguibile non prende il controllo la funzione **main**
- **_start** esegue task preliminari (funzioni di ambiente) che permettono al **main** di eseguire correttamente
  - Passare parametri e variabili di ambiente al main

# Layout di un programma C



# Variabili d’ambiente

- PWD directory corrente
- HOME directory principale dell’utente
- PATH specifica per la ricerca di eseguibili
- Funzione di gestione:
  - getenv
  - putenv
  - setenv
  - unsetenv
- E la fork?
  - Le variabili di ambiente vengono ereditate dal processo figlio

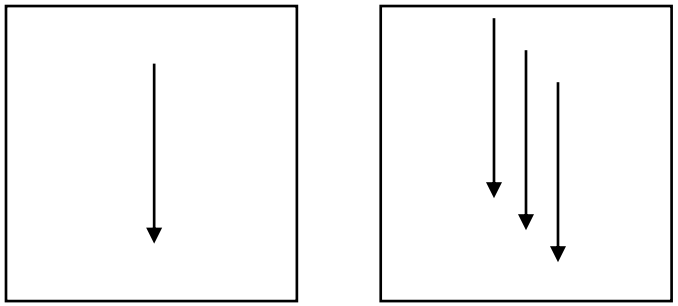
# (Alcuni) Servizi di sistema per gestire i processi

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

void main() {
 char comando[256];
 pid_t pid; int status;

 while(1) {
   printf("Digitare un comando: ");
   scanf("%s",comando);
   pid = fork();
   if ( pid == -1 ) {
     printf("Errore nella fork.\n");
     exit(1);
   }
   if ( pid == 0 )
     execlp(comando, comando, NULL);
   else wait(&status);
   }
}
```

# Processi e threads



- Il processo ingloba
  - Risorse assegnate
  - Traccia di istruzioni
- I due concetti possono essere disaccoppiati
  - Medesime risorse assegnate
  - Più tracce di istruzioni
- L'unità di base per il dispatching è il thread
- L'unità proprietaria delle risorse è il processo
- Più thread possono appartenere al medesimo processo
- **ATTENZIONE!** i thread condividono le risorse
  - **Memoria inclusa**

# Processi e threads

- Un programma
- I dati su cui opera
- Almeno uno stack
- Dati di contesto
- Rif. risorse hardware
- Identificativi
- Statistiche
- Uno stato

PCB1

Immagine di processo P1

- Un programma
- I dati su cui opera

- Almeno uno stack
- Dati di contesto
- Identificativi
- Statistiche
- Stato

TCB

Thread A

- Almeno uno stack
- Dati di contesto
- Identificativi
- Statistiche
- Stato

TCB

Thread B

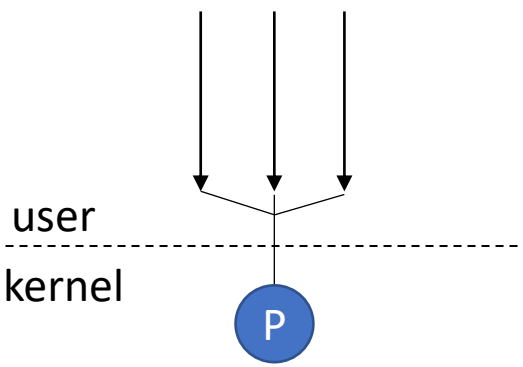
- Rif. risorse hardware
- Identificativi
- Statistiche
- Uno stato

PCB1

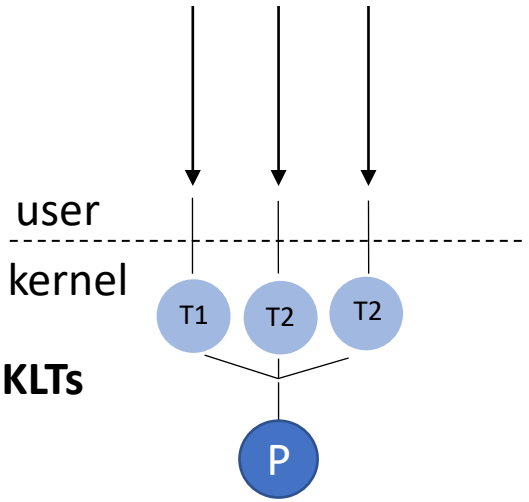
Immagine di processo P1

# Processi e threads – un'altra prospettiva

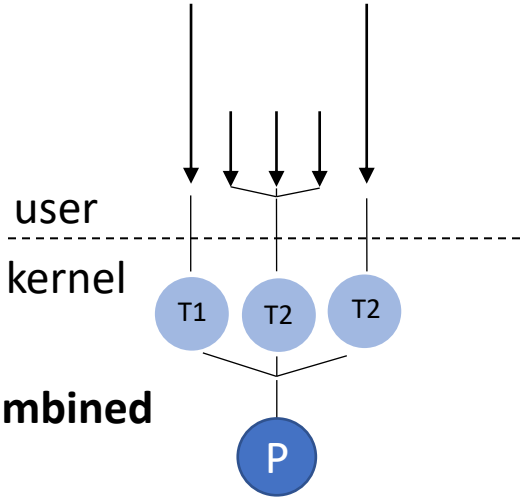
- **User-Level Threads (ULTs) :**
  - La gestione di più flussi di controllo è fatta interamente in modo user
  - Il sistema operativo non è a conoscenza dell'esistenza dei thread
    - L'unità per il dispatch è ancora il processo
- **Kernel-Level Threads (KLT):**
  - Il sistema operativo ha tutti i supporti necessari alla gestione dei threads
    - Mantiene informazioni per il processo e per i suoi thread
    - L'unità per il dispatch è il thread



ULTs



KLTs



Combined

# Gestione di thread

- La libreria ULT o il sistema operativo offrono dei servizi per:
  - Creare thread
  - Terminare l'esecuzione del thread corrente
  - Aspettare che uno specifico thread termini la sua esecuzione
- Standard POSIX:
  - pthread_create
  - pthread_exit
  - pthread_join
- E la exit in ambienti multi-threaded?
  - è tipicamente mappata su un'altra system call (e.g., exit_group)
  - La system call exit termina solo il thread corrente

# Variabili per thread

- Attraverso appositi costrutti è possibile definire variabili **globali** associate ad ogni **thread**
  - Esiste un'istanza della variabile per ciascun thread
- Variabili thread local definite a tempo di compilazione
  - **__thread** nella toolchain di compilazione GNU (gcc)
- Variabile definite a runtime (POSIX)
  - pthread_key_create
  - pthread_key_delete
  - pthread_get_specific
  - pthread_set_specific
- Ogni thread può accedere la variabile con l'apposito simbolo o puntatore

# Multithreading e librerie

- Una libreria viene definita **thread safe** se le sue funzioni supportano invocazioni concorrenti da più thread
  - la gestione interna dello stato supporta l'esecuzione concorrente di  funzioni di libreria da parte di più thread
- Diverse librerie (o funzioni) sono thread safe per default
  - printf
  - malloc
- VERIFICARE **SEMPRE** LA THREAD SAFETY CONSULTANDO LA DOCUMENTAZIONE

# E il kernel?

- I kernel hanno utilizzato tecnologie multi-thread ancor prima di renderle disponibili a sviluppatori applicativi
- Esistevano/esistono «thread» concorrenti privi di immagine (e quindi programma) user-level (chiamati anche kernel threads – da non confondere con il concetto di lightweight process o KLT)
- I kernel threads condividono la stessa immagine (quindi dati e risorse) dello stesso programma (kernel di sistema operativo)

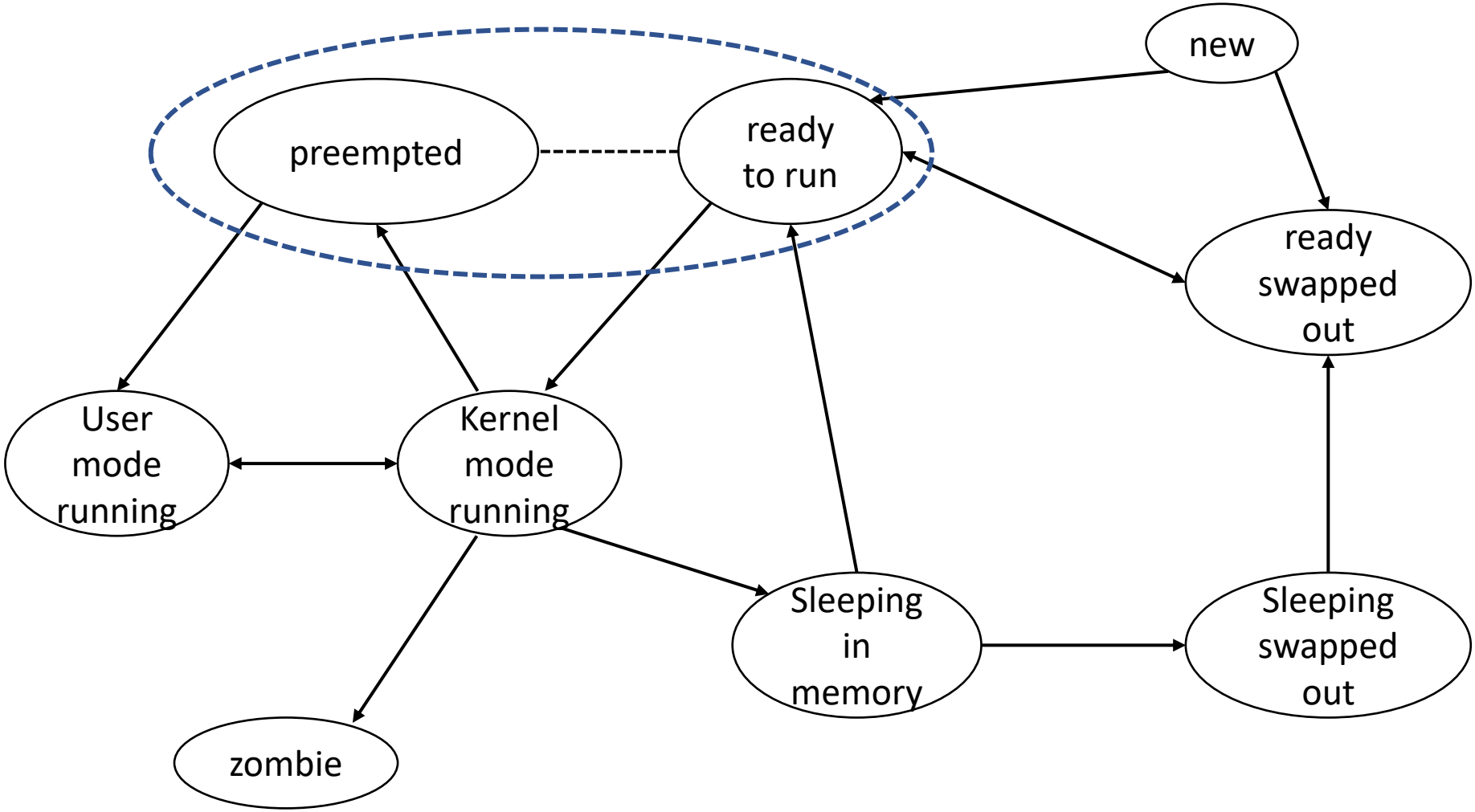
# (Alcuni) Servizi di per gestire i thread

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <pthread.h>

void main(){
    pthread_t ctid;
    int res, *status_ptr, status_val;
    status_ptr = &status_val;
    printf("I'm a thread in a process. "
           "I'm going to create a thread\n");
    res = pthread_create(&ctid, NULL, child_func, status_ptr);
    if(res != 0) printf("I cannot create a child");
    else{
        printf("I'm now a parent thread. "
               "I'll wait for my child to die...\n");
        pthread_join(ctid, (void*)&status_ptr);
        printf("My child has completed...%d\n",*status_ptr);
    }
    printf("My child is dead, so it's my time to die\n");
    exit(0);
}
```

```
void* child_func(void *par){
  *((int*)par) = 1;
  sleep(10);
  printf("I'm the child!\n");
  pthread_exit(par);
}
```

# Processi – UNIX



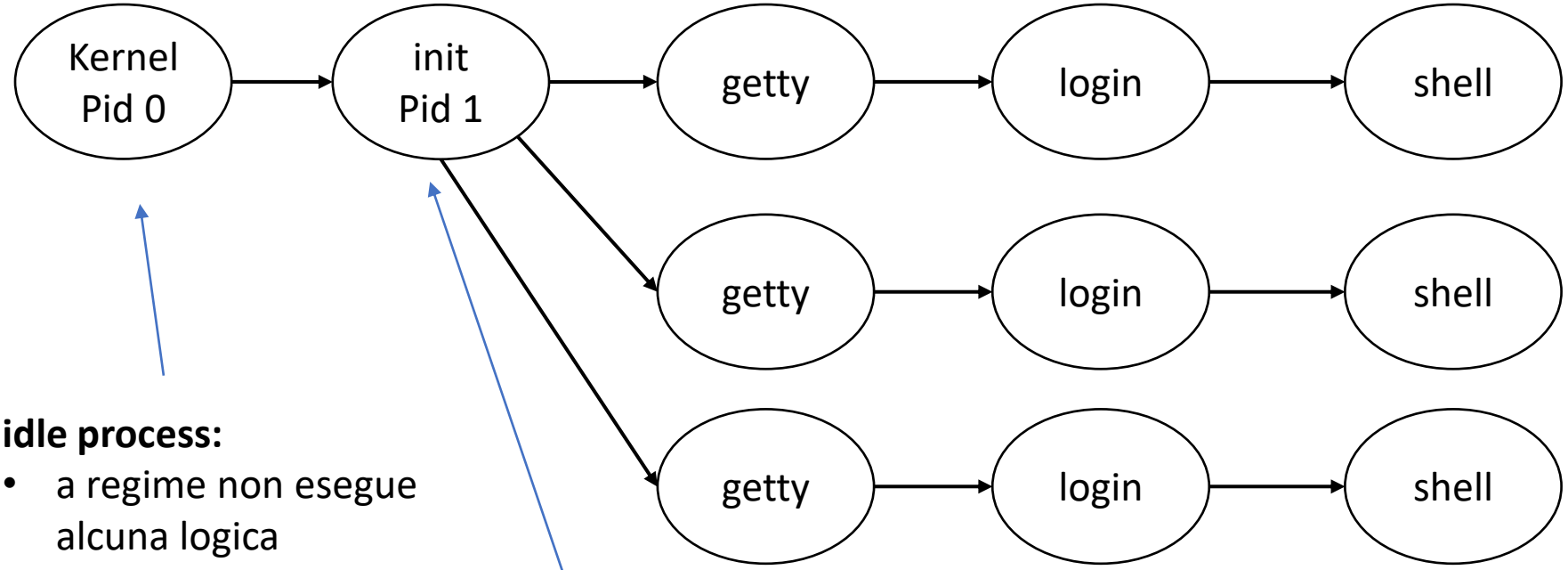
# Immagine di processo - UNIX

- Contesto utente
  - Testo
  - Dati
  - Stack user
  - Memoria condivisa
- Contesto registri
  - Program counter
  - Registri di stato del processore
  - Stack pointer
  - Registri general purpose
- Contesto sistema
  - Entry nella tabella dei processi
  - Stack kernel
  - U area
  - Tabelle di indirizzamento

# Immagine di processo - UNIX

- PCB
  - Stato del processo
  - Identificatori di processo
  - Affinità di processore
  - Priorità
  - Timer
  - Stato della memoria
  - Segnali
- U area
  - Identificatori d'utente
  - Gestori di segnali
  - Terminale
  - Tabella dei descrittori di file
  - Valori di ritorno di system call

# Gerarchie di processi – UNIX




**idle process:**

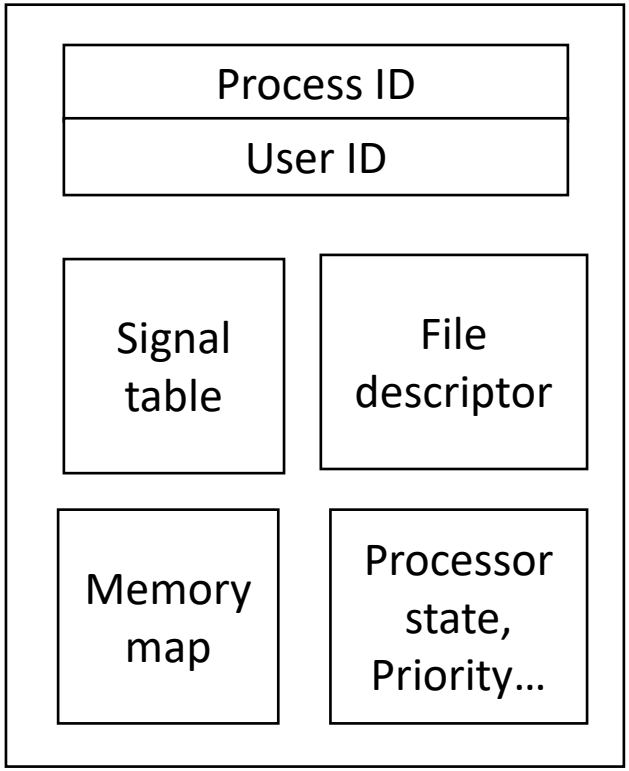
- a regime non esegue alcuna logica
- se non richiedere l'attivazione dello scheduler.
- esegue istruzioni per ridurre il consumo energetico (e.g., MONITOR, MWAIT su x86)

- Legge alcuni file di configurazione ed avvia servizi (ad esempio per accettare login remoti e/o locali)
- Anche chiamato **systemd**

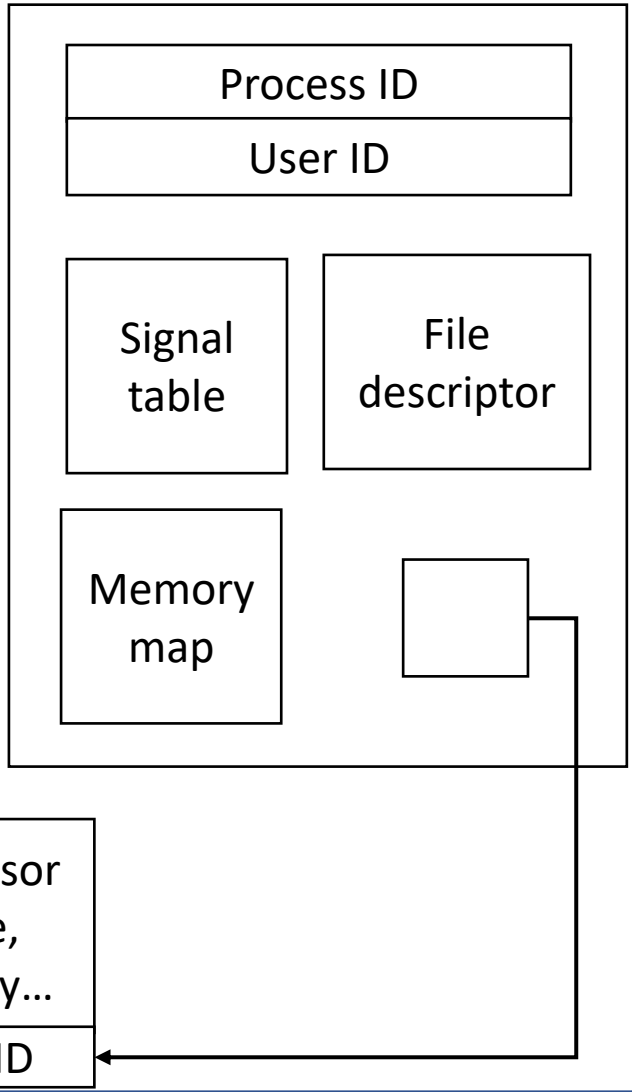
- Legge il file relativo alle password /etc/passwd

# Processi e thread – UNIX (Solaris)

Traditional UNIX process structure

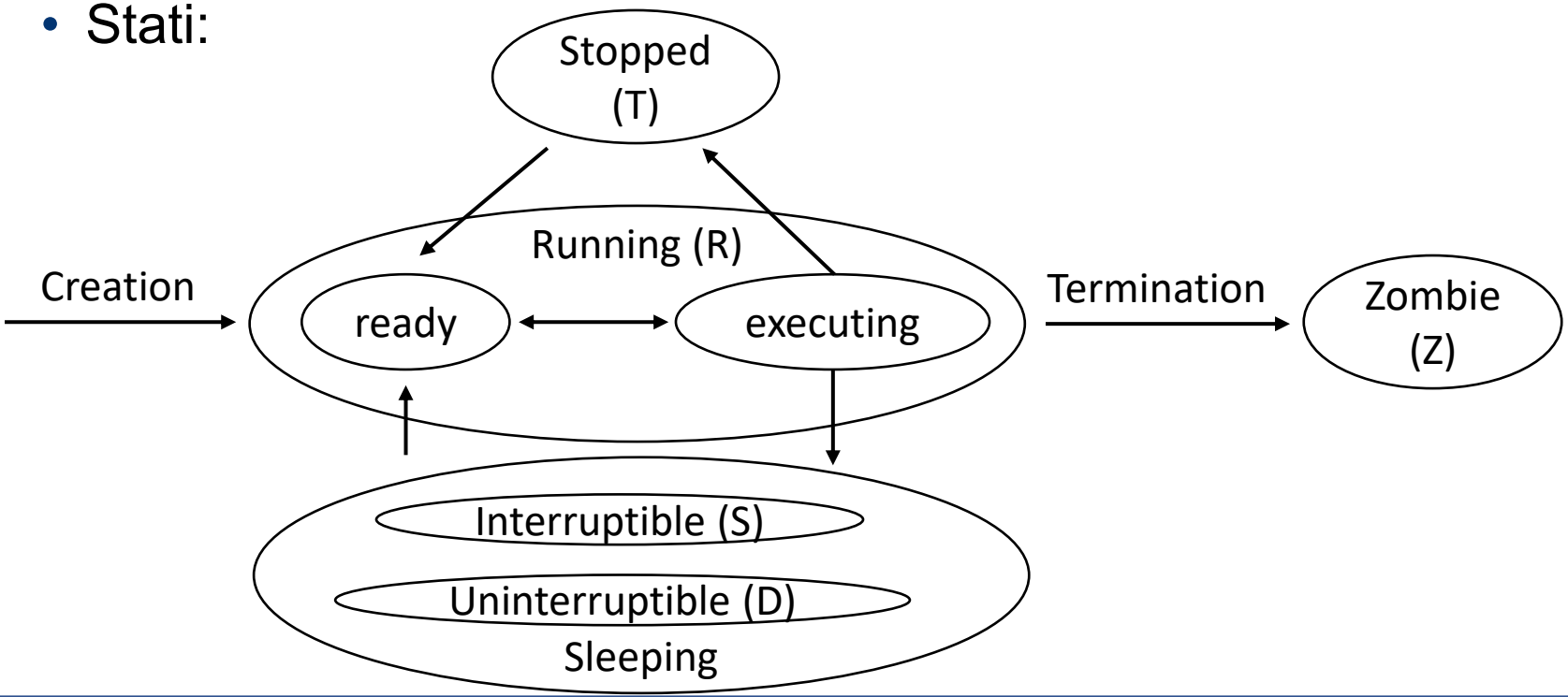



Solaris process structure



# Processi e thread – UNIX (Linux)

- Non esiste una reale distinzione tra thread e processi
- Esistono i task
- Il PCB è chiamato task struct
  - https://elixir.bootlin.com/linux/v5.14.7/source/include/linux/sched.h#L661
- Stati:



# Processi e thread – UNIX (Linux)

- Come mappano i concetti di processo e thread sui task?
  - Thread = task che **condividono** lo spazio di indirizzamento
  - Processi = task che **non condividono** lo spazio di indirizzamento
- Linux espone un servizio di sistema per creare un nuovo task e configurare gli elementi condivisi con il parent task
  - Syscall clone
- Attraverso alcuni flag è possibile definire cosa condividono i due task:
  - CLONE_VM: condividere lo spazio di indirizzamento
  - CLONE_FILES: condividere la tabella relativa alla gestione dei file
- In Linux, la libreria pthread utilizza la clone per creare un nuovo thread

# Funzione/librerie rientranti

- Una funzione è rientrante se può essere ri-eseguita dal medesimo processo prima che l'invocazione corrente sia terminata
- Funzione rientrante