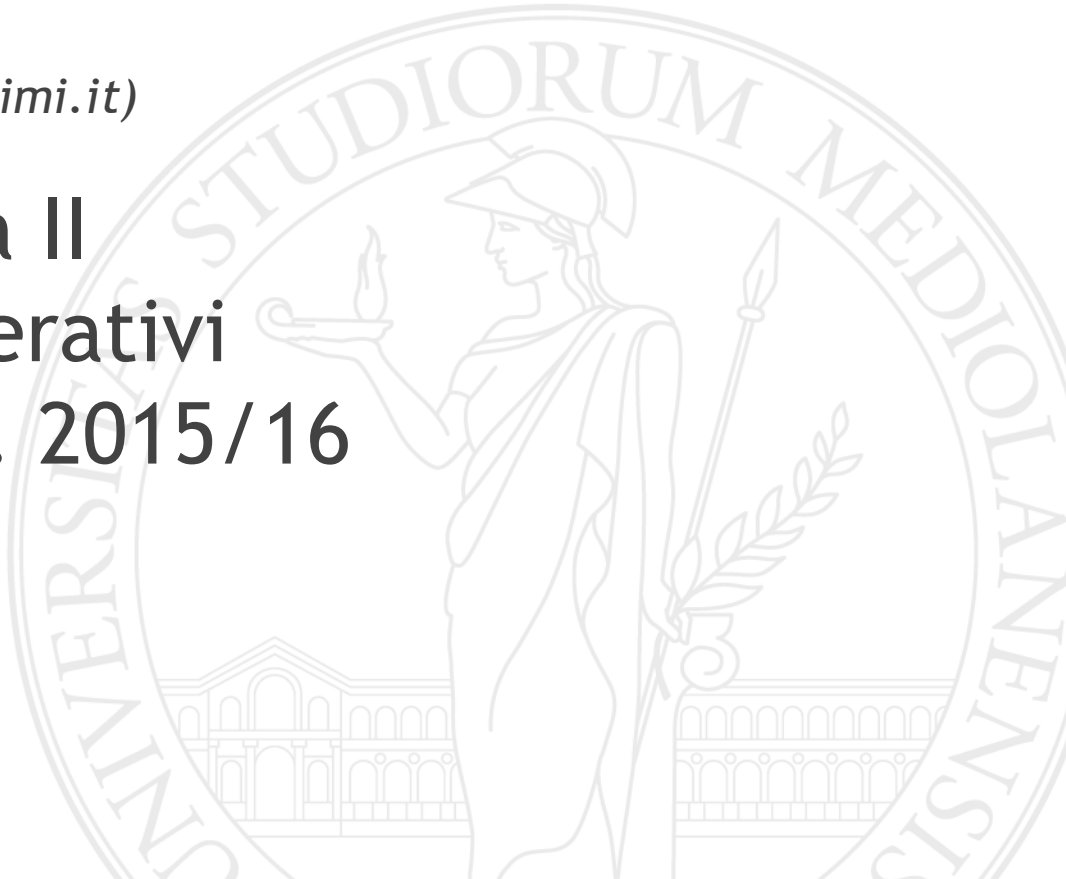
UNIVERSITÀ DEGLI STUDI DI MILANO
DIPARTIMENTO DI INFORMATICA

Alberto Ceselli
(alberto.ceselli@unimi.it)

Informatica II
Sistemi Operativi
DIGIP - a.a. 2015/16



Sistemi Operativi

(modulo di Informatica II)

I processi

Patrizia Scandurra

Università degli Studi di Bergamo

Sommario

- Il concetto di processo
- Schedulazione dei processi e cambio di contesto
- Operazioni sui processi

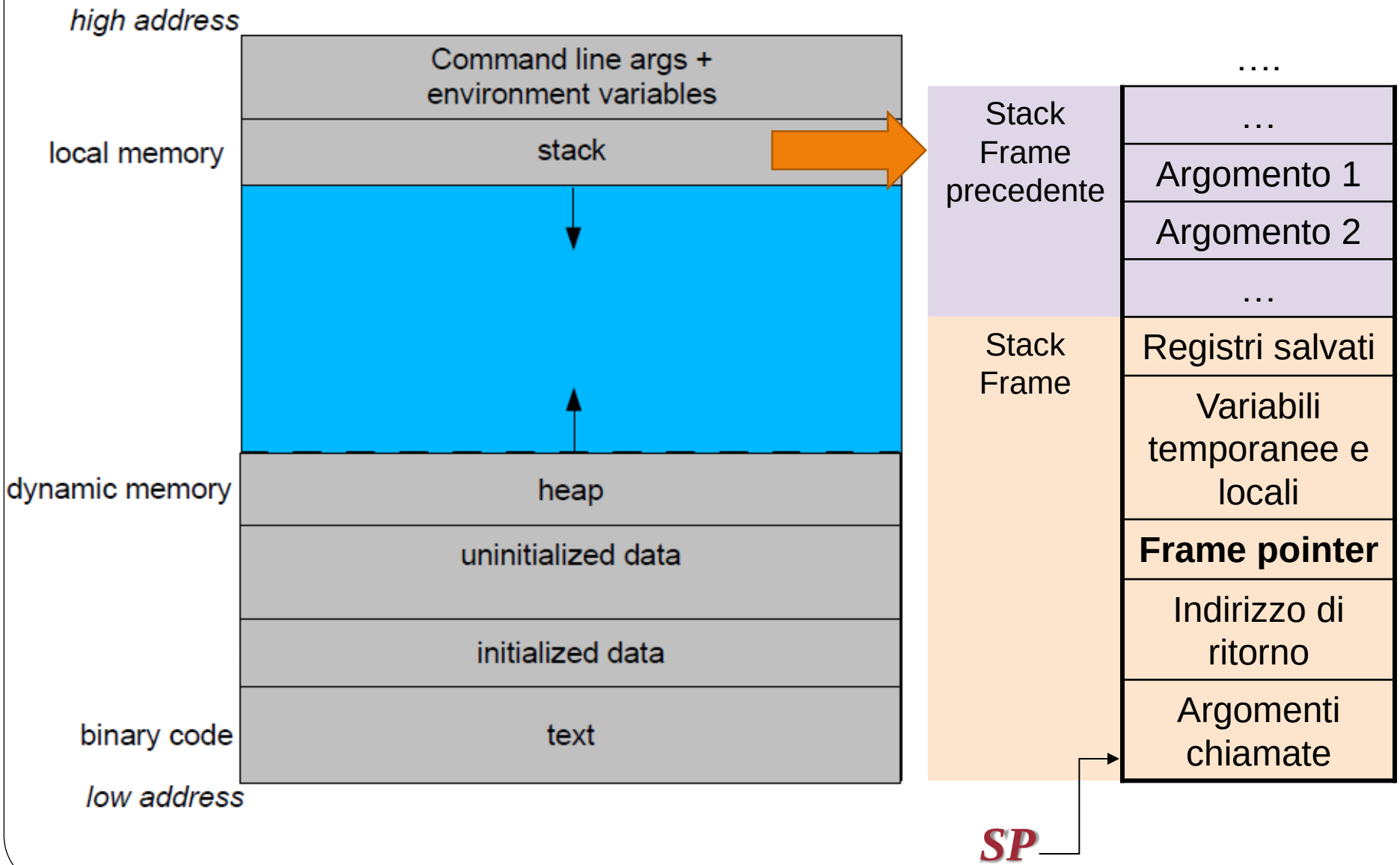
Il concetto di processo (1)

- Un SO esegue una varietà di programmi:
  - Sistema a lotti (batch system) – lavoro (jobs)
  - Sistema a condivisione di tempo (time-sharing) – programmi utente (tasks)
- I termini **job** e **processo** spesso usati come sinonimi
- ***Processo*** – "*un programma in esecuzione*"
  - Codice
  - Dati
  - Flusso di controllo

Il concetto di processo (2)

- *Componenti*:
  - **Codice** del programma (text o *code section*)
  - **Dati** del programma
    - Variabili globali in memoria centrale (*data section*)
    - Variabili locali e non locali (*stack*)
    - Variabili temporanee generate dal compilatore (registri del processore)
    - Variabili allocate dinamicamente (*heap*)
      - anche se l'allocazione e la deallocazione della memoria all'interno di questo segmento sono a carico del programmatore
  - **Stato di evoluzione della computazione**
    - Program counter
    - Valore delle variabili
    - Informazioni di gestione del contesto della chiamata di procedure (indirizzo di ritorno, frame pointer, stack pointer)

Memory layout di un processo



Programma ≠ Processo

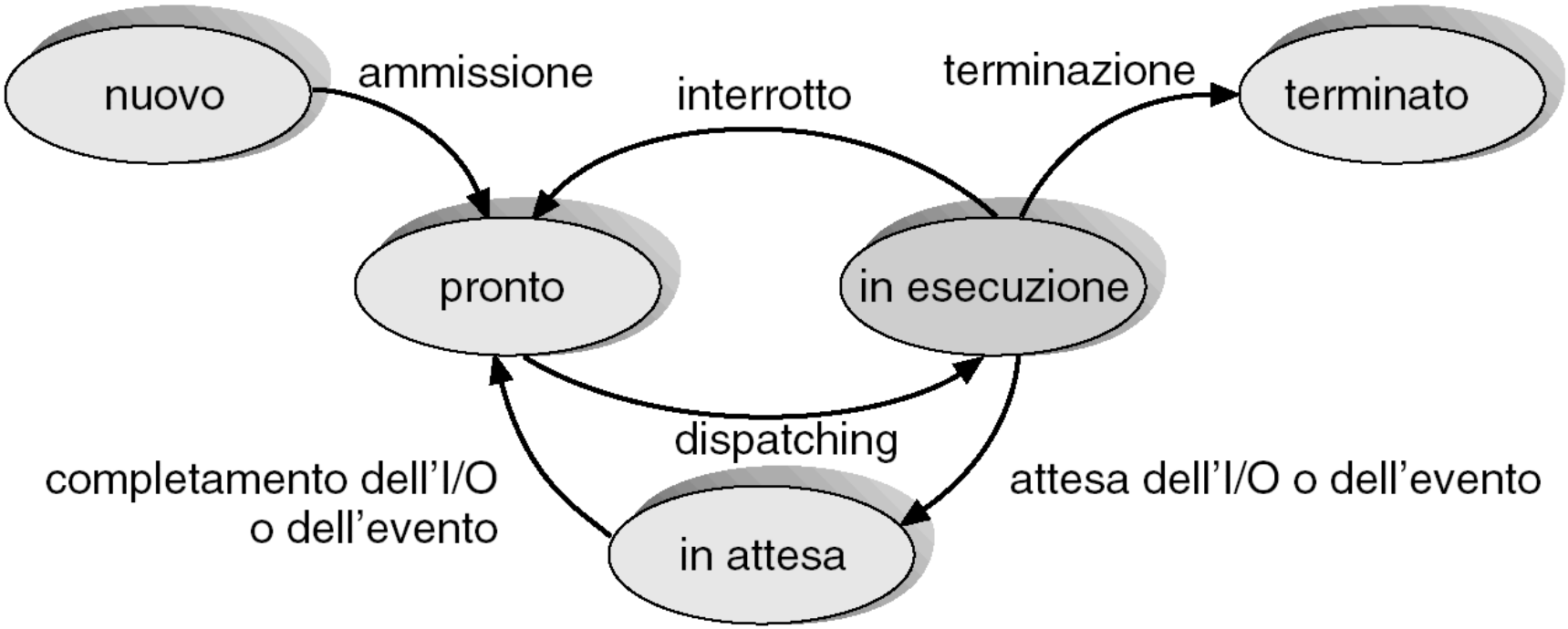
- **Programma**
  - Entità passiva
  - Lista istruzioni
- **Processo**
  - Entità attiva
  - Valori delle variabili
  - Risorse in uso
- Anche se due processi possono essere associati allo stesso programma, essi sono due differenti *istanze di esecuzione* dello stesso codice!

Lo stato di un processo

- E' lo **stato di uso del processore** da parte di un processo

- Possibili stati:
  - Nuovo (new): Il processo è stato creato
  - In esecuzione (running): le istruzioni vengono eseguite
  - In attesa (waiting): il processo sta aspettando il verificarsi di qualche evento
  - Pronto all'esecuzione (ready): il processo è in attesa di essere assegnato ad un processore
  - Terminato (terminated): il processo ha terminato l'esecuzione

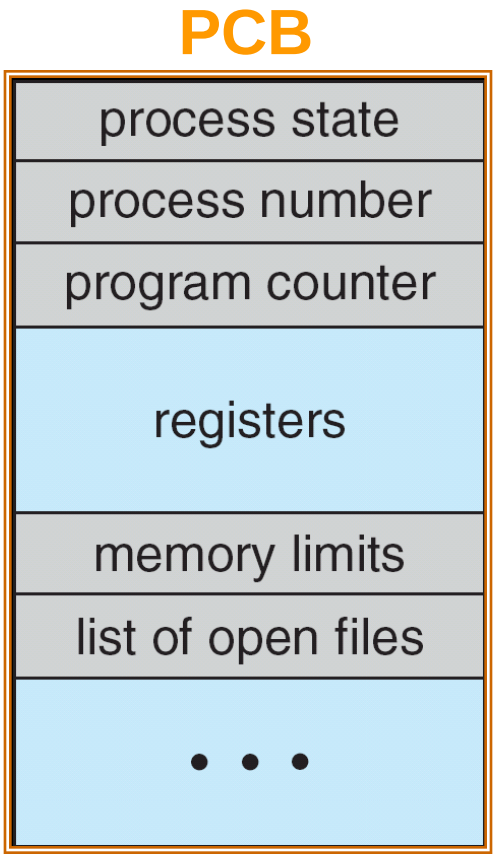
Diagramma degli stati di un processo



Supporti per la gestione dei processi (1)

Process Control Block (PCB)

- Struttura dati del kernel che mantiene le informazioni sul processo
  - Stato del processo
  - Identificatore del processo (Numero)
  - Program counter
  - Registri della CPU
  - Info per la gestione della memoria centrale (limiti di memoria)
  - Info sullo stato dell'I/O (ad es.: file aperti)
  - Info per la schedulazione della CPU
  - Info per l'accounting ecc…



PCB di Linux (vers. 0.01 dal file include/linux/sched.h)

```
struct task_struct {
        long state;     /* -1 unrunnable, 0 runnable, >0 stopped */
        long counter;
        long priority;
        long signal;
        fn_ptr sig_restorer;
        fn_ptr sig_fn[32];
/* various fields */
        int exit_code;
        unsigned long end_code,end_data,brk,start_stack;
        long pid,father,pgrp,session,leader;
        unsigned short uid,euid,suid;
        unsigned short gid,egid,sgid;
        long alarm;
        long utime,stime,cutime,cstime,start_time;
        unsigned short used_math;
        ...
```

PCB di Linux (vers. 0.01 dal file include/linux/sched.h) (cont.)

```
        ...
/* file system info */
        int tty;                /* -1 if no tty, so it must be signed */
        unsigned short umask;
        struct m_inode * pwd;
        struct m_inode * root;
        unsigned long close_on_exec;
        struct file * filp[NR_OPEN];
/* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
        struct desc_struct ldt[3];
/* tss for this task */
        struct tss_struct tss;
};
```

27 righe commenti inclusi; sono 134 in Linux 2.4.18

416 in un kernel linux 3.13

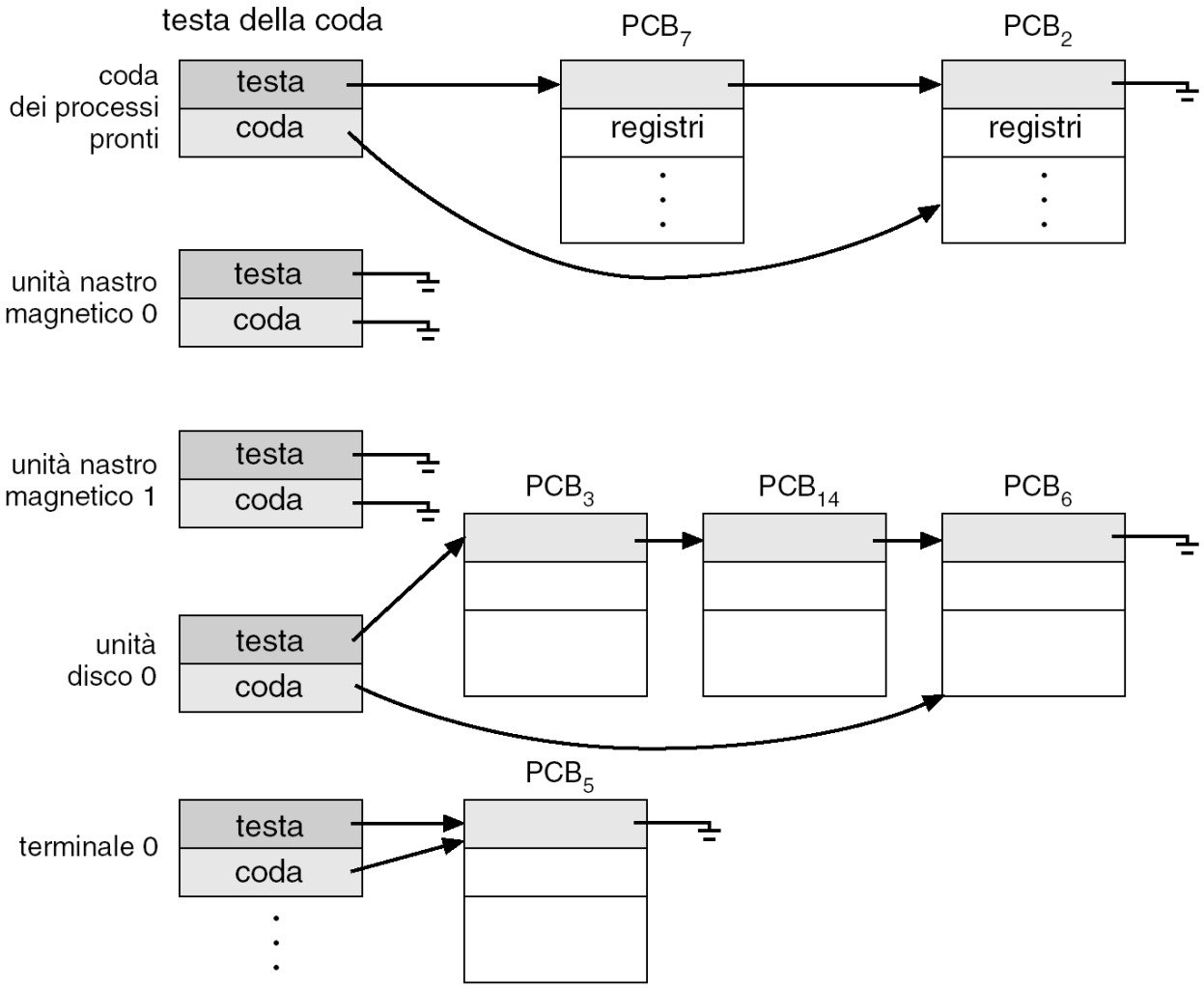
Supporti per la gestione dei processi (2)

Le code di schedulazione dei processi

- Coda di **lavori** *(job queue)* – *contiene tutti i processi nel sistema*
- Coda dei **processi pronti** *(ready queue)* – *contiene tutti i processi* che risiedono nella memoria centrale, pronti e in attesa di esecuzione
- Coda della **periferica di I/O** *(device queues)* – *contiene i processi* in attesa di una particolare periferica di I/O
- Il processo si muove fra le varie code

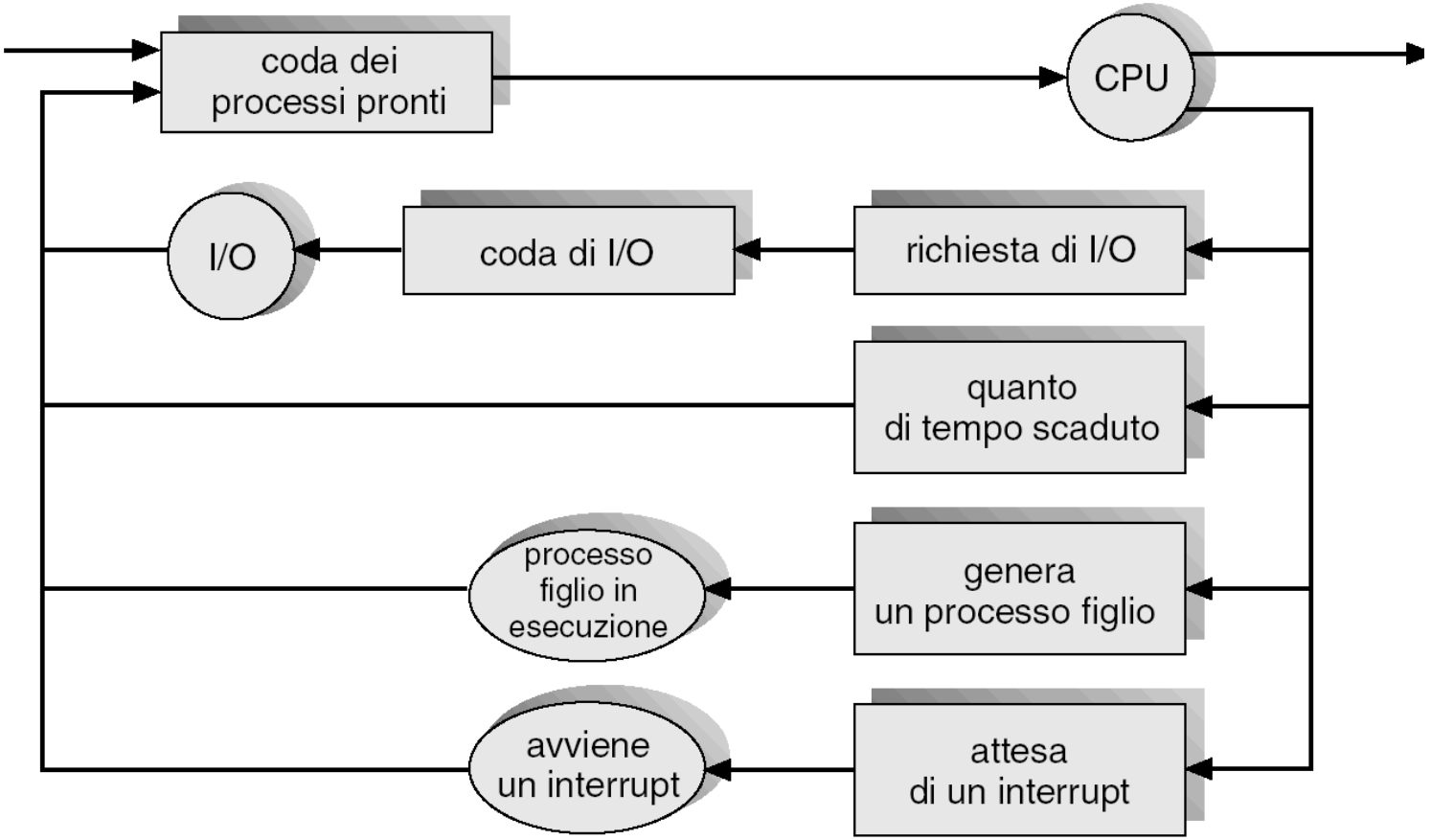
Supporti per la gestione dei processi (2)

Code dei processi nei vari stati



Supporti per la gestione dei processi (3)

Transizioni tra le code



Gli schedulatori (1)

- ***Schedulatore a lungo termine*** *(long-term scheduler)* – seleziona quale processo (attualmente in memoria di massa) deve essere inserito nella coda dei processi pronti
  - Controlla il *grado di multi-programmazione*
  - Richiamato solo quando un processo abbandona il sistema (termina)
- ***Schedulatore a breve termine*** *(Short-term scheduler)* – seleziona *quale* processo (in memoria) deve essere eseguito e alloca la CPU ad esso
- ***Schedulatore a medio termine*** *(Medium-term scheduler)* – *SWAPPING* rimuove un processo dalla memoria centrale e lo pone in memoria di massa
  - Supportato solo da alcuni SO come i *sistemi time-sharing*

Gli schedulatori (2)

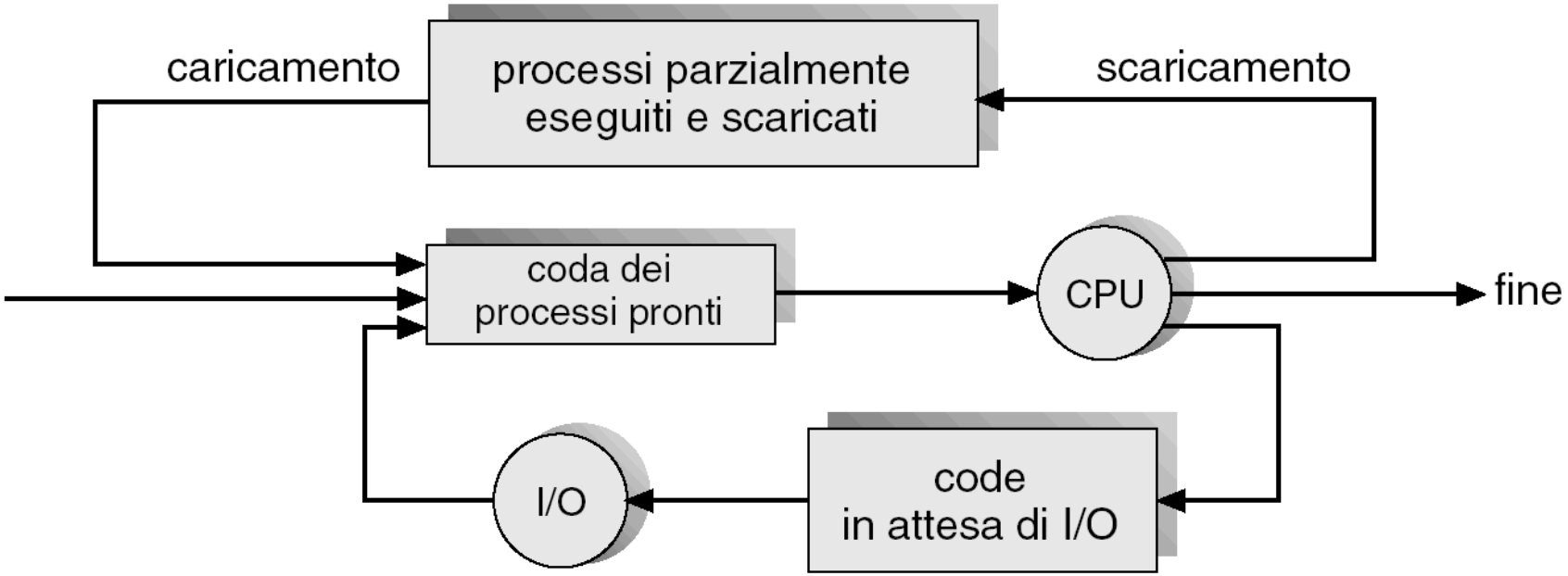


Diagramma delle code con aggiunta dello schedulatore a medio termine

→Rimuove processi dalla memoria centrale: swapping out
→Reintroduce in memoria centrale i processi: swapping in

Gli schedulatori (3)

- **Lo schedulatore a breve termine** è eseguito molto frequentemente (millisecondi) ⇒ (deve essere veloce)

- **Lo schedulatore a lungo termine** è eseguito molto meno frequentemente (secondi, minuti) ⇒ (può essere lento)
  - In alcuni **SO può essere assente** (ad es. in sistemi time-sharing, come Unix e MSWindows, il grado di multi-programmazione è regolato dallo schedulatore a medio termine)
  - Ci si limita a caricare **tutti i nuovi processi in memoria**

Processi CPU-bound e I/O-bound

- I processi possono essere classificati come:
  - **processo I/O-bound** – processo che spende più tempo facendo I/O che elaborazione
    - molti e brevi utilizzi di CPU
  - **processo CPU-bound** – processo che spende più tempo facendo elaborazione che I/O
    - pochi e lunghi utilizzi di CPU
- Un sistema di buone prestazioni presenta una combinazione bilanciata di processi CPU-bound e I/O-bound
- Lo swapping può essere necessario per migliorare la distribuzione dei processi tra le due tipologie
  - oppure perché un cambiamento di richieste della memoria centrale ha superato la memoria disponibile

Il cambio di contesto (1)

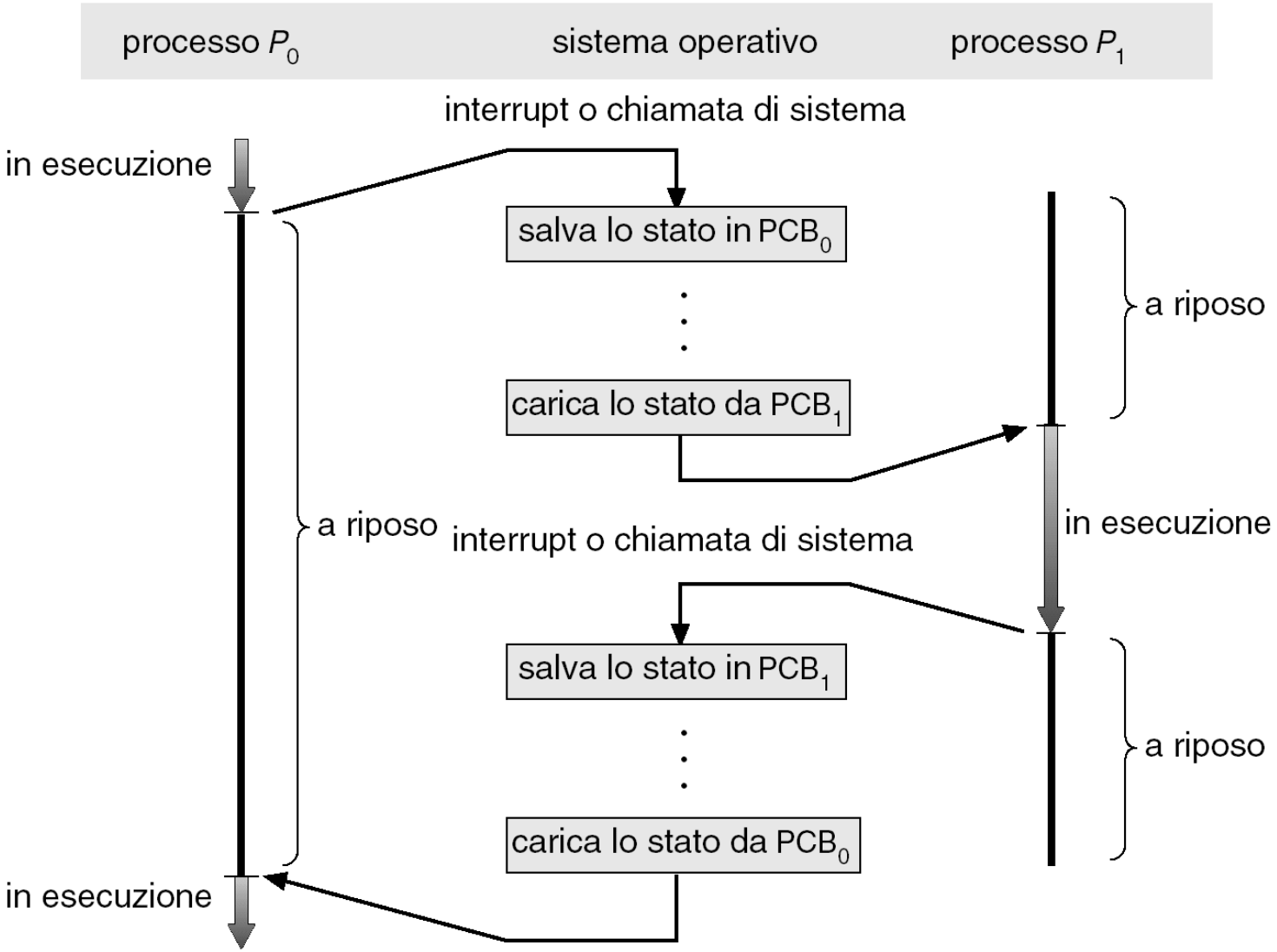
context switch

- E' il **passaggio della CPU da un processo ad un altro**

Context Switch = *sospensione del processo in esecuzione* + *caricamento del nuovo processo da mettere in esecuzione*

- Il *dispatcher* è il modulo che passa effettivamente il controllo della CPU ai processi scelti dallo scheduler a breve termine
- Il tempo per il cambio di contesto (*latenza di dispatch*) è puro **tempo di gestione del sistema**
  - poichè durante il cambio non vengono compiute operazioni utili per la computazione dei processi
- I tempi per i cambi di contesto dipendono sensibilmente dal supporto hardware
  - tipicamente inferiore ai 10 millisecondi

Il cambio di contesto (2)



Processi come flussi di operazioni

- Processo = flusso di esecuzione di computazione

- Flussi separati = processi separati
  - Come si generano?
  - Come un programma si trasforma in un insieme di processi?
  - Come interagiscono i processi?

- Flussi sincronizzati

  → processi evolvono sincronizzandosi
- Flussi indipendenti

  → processi evolvono autonomamente

Modellazione della computazione a processi

- Modelli di computazione:
  - Processo monolitico
  - Processi cooperanti
- Modelli di realizzazione del codice eseguibile:
  - Programma monolitico
  - Programmi separati
- Realizzazione dei modelli di computazione:
  - Programma monolitico è eseguito come processo monolitico
  - Programma monolitico genera processi cooperanti
  - Programmi separati sono eseguiti come processi cooperanti (ed eventualmente generano ulteriori processi cooperanti)

Creazione (o generazione) di un processo

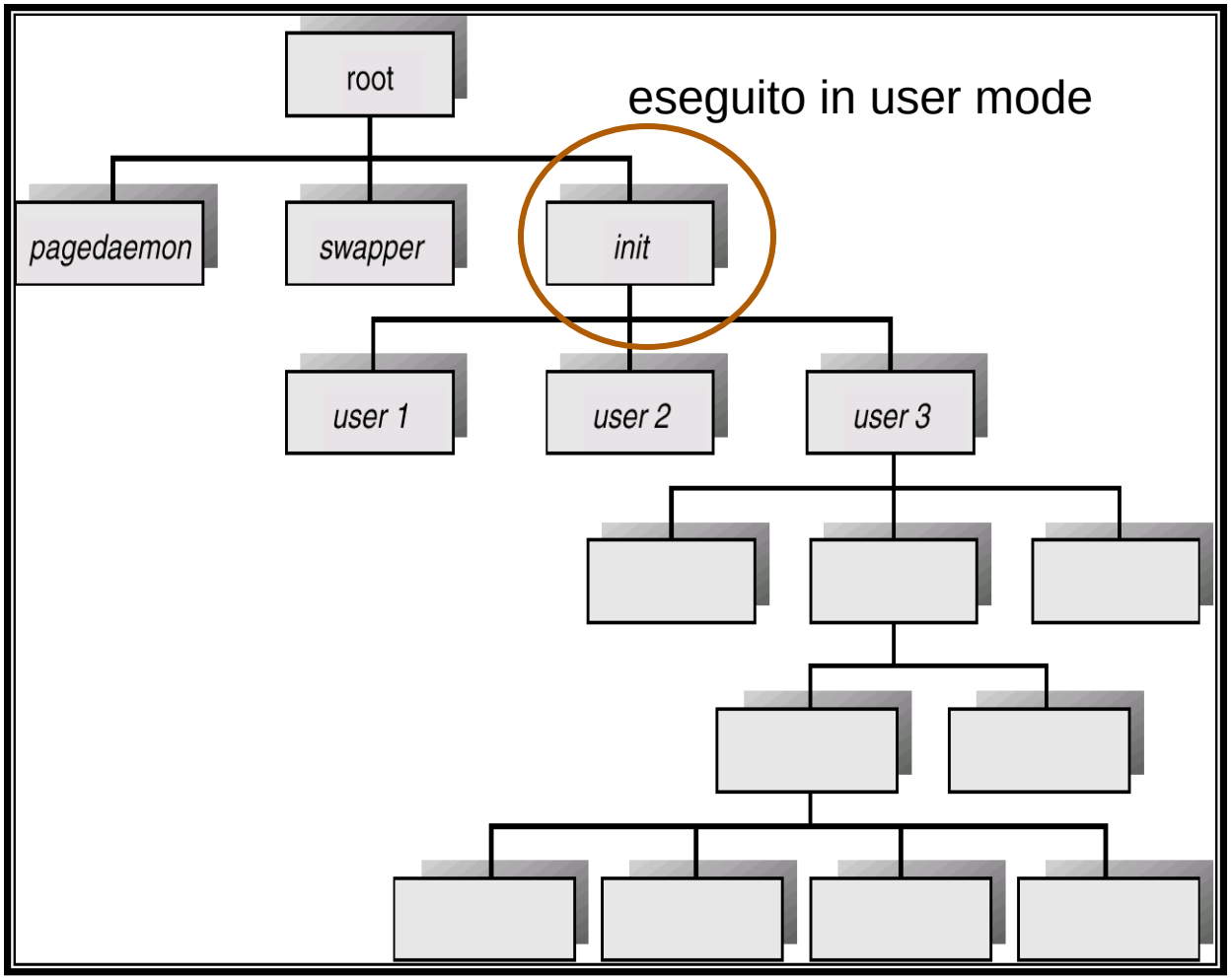
- Il processo in esecuzione invoca una chiamata di sistema che crea e attiva un nuovo processo
  - Ad es. in SO Unix-like la chiamata *fork()*

  - Processo generante → processo padre
  - Processo generato → processo figlio

Albero dei processi

Il processo padre crea processi figli, i quali a loro volta creano altri processi formando un albero di processi

Es. albero dei processi in un sistema Unix-like



Risorse dei processi

Un nuovo processo può ottenere risorse in diversi modi:

- Condivise col padre
- Parzialmente condivise col padre
- Indipendenti dal padre (ottenute dal sistema)
  - Limitare le risorse del figlio ad un sottoinsieme di quelle del padre aiuta ad evitare che un processo crei troppi figli

Inoltre alla creazione un processo figlio riceve dal padre i dati di inizializzazione

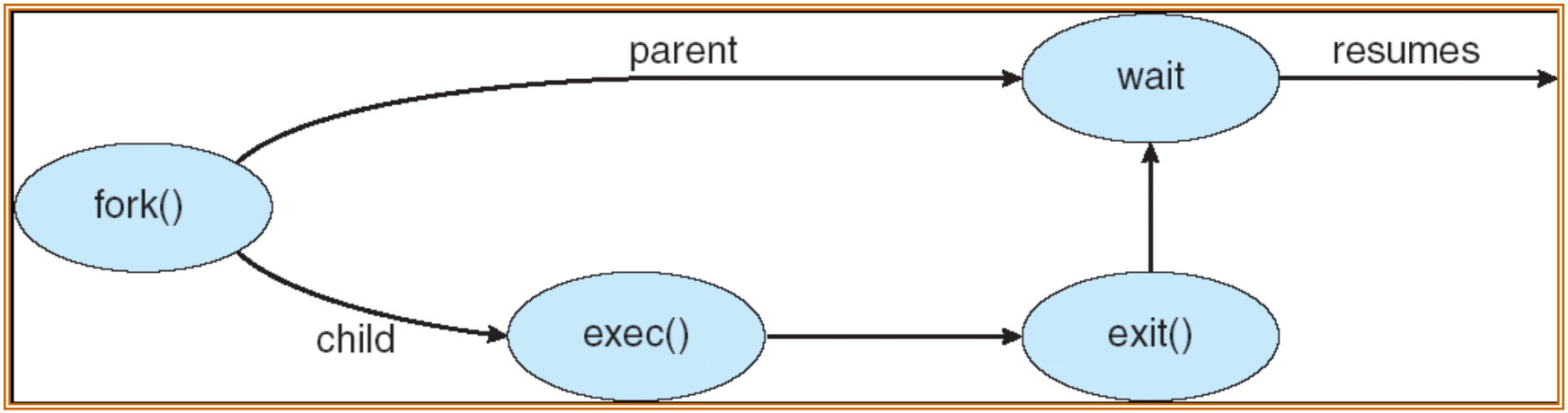
- Parametri utili per l'esecuzione del processo
  - Ad es. un processo che permette di visualizzare un immagine riceverà dal padre il percorso del file

Spazio di indirizzamento (1)

- Lo spazio di indirizzamento (memoria) del processo figlio è sempre **distinto** da quello del processo padre
- Due possibili scenari:
  - Il figlio è un **duplicato del padre**
    - stesso programma
    - stessi dati all'atto della creazione
  - Il figlio ha un **nuovo programma** caricato nel proprio spazio di indirizzamento
- In Unix la chiamata fork() crea un duplicato del padre
  - Lo spazio degli indirizzi può essere successivamente sovrascritto

Esecuzione dei processi

- Due scenari possibili:
  - Il padre continua l'esecuzione in modo concorrente ai figli
    - modalità asincrona
  - Il padre attende finchè tutti (o alcuni) i suoi figli sono terminati
    - modalità sincrona



Esempio in UNIX

La chiamata di sistema *exec, usata dopo la fork, carica un nuovo programma* nello spazio di memoria del processo che la esegue

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
int pid;

    /* genera un altro processo */
    pid = fork();

    if (pid < 0) { /* si è verificato un errore */
        fprintf(stderr, "Fork fallita");
        exit(-1);
    }
    else if (pid == 0) { /*  processo figlio */
        execlp("/bin/ls","ls",NULL);
    }
```

Terminazione di un processo (1)

- Terminazione normale dopo l'ultima istruzione tramite la chiamata *exit*
  - "figlio" può restituire un **valore di stato** (di solito un intero) al "padre"
    - Nell'es. tramite la chiamata *wait()*
  - le risorse del processo sono deallocate dal SO

```
    }
    else if (pid == 0) { /*  processo figlio */
        execlp("/bin/ls","ls",NULL);
    }
    else { /* processo padre */
        /* il processo padre attenderà il completamento del figlio */
        wait(NULL);
        printf("Figlio terminato");
        exit(0);
    }
}
```

In Windows

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    // alloca la memoria
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // genera processo figlio
    if (!CreateProcess(NULL, // usa riga di comando
     "C:\\WINDOWS\\system32\\mspaint.exe", // riga di comando
     NULL, // non eredita l'handle del processo
     NULL, // non eredita l'handle del thread
     FALSE, // disattiva l'ereditarieta' degli handle
     0, // nessun flag di creazione
     NULL, // usa il blocco ambiente del genitore
     NULL, // usa la directory esistente del genitore
     &si,
     &pi))
    {
      fprintf(stderr, "generazione del nuovo processo fallita");
      return -1
    }
    // il genitore attende il completamento del figlio
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("il processo figlio ha terminato");

    // rilascia gli handle
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

- STARTUPINFO e PROCESS_INFORMATION sono strutture dati contenenti info sui processi
- ZeroMemory alloca la memoria per il nuovo processo
- CreateProcess() equivale alla fork()
  - Riceve molti parametri
- WaitForSingleObject() è il corrispettivo di wait()
  - Riceve come parametro il processo da attendere

Terminazione di un processo (2)

- Terminazione in caso di anomalia (aborto)

- Il padre può terminare l'esecuzione di uno dei suoi figli per varie ragioni:
  - Eccessivo uso di una risorsa
  - Compito non più necessario
  - Terminazione a cascata
    - se il padre sta terminando, alcuni SO non permettono ad un processo figlio di proseguire

Esempi di terminazione in C

normale:

```
main () {
   return 0;
}
```

per errore: (qualsiasi numero != 0)

```
main () {
   return 1;
}
```

oppure:

```
void f() {
    _exit(0);
}
main () {
   f();
}
```

oppure:

```
void f() {
    _exit(1);
}
main () {
   f();
}
```

Con uno stato non determinato:

```
main () { }
```