Sistemi Operativi Modulo I

Secondo canale (M-Z)

A.A. 2021/2022

Corso di Laurea in Informatica

4. Processi

Paolo Ottolino

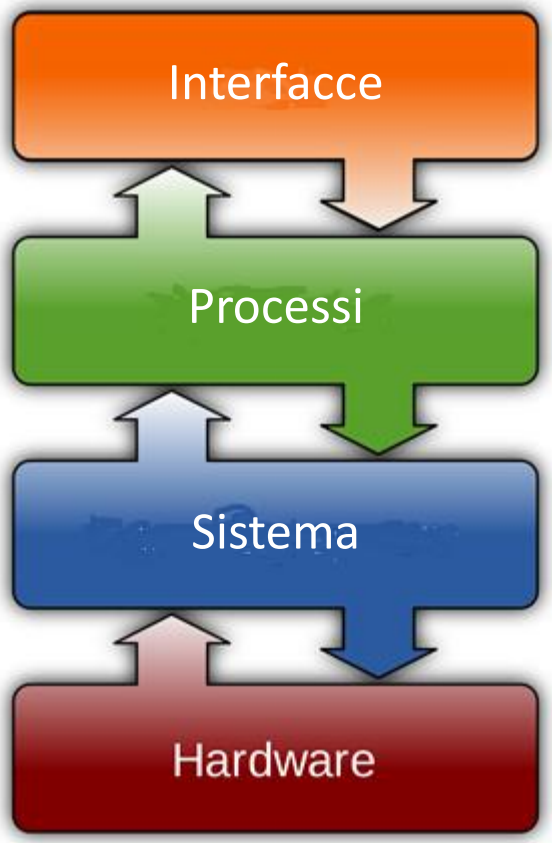
Sapienza Università di Roma

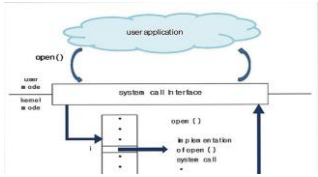
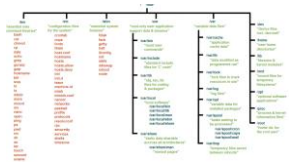
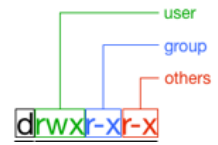
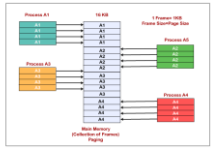
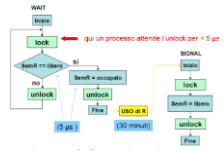
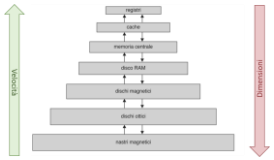
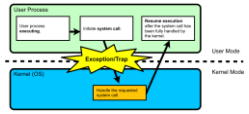
Dipartimento di Informatica

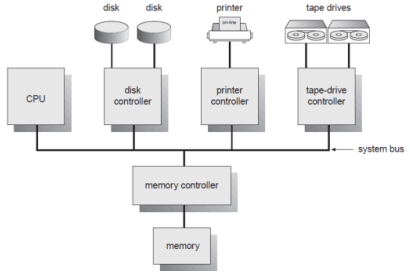
Operating Systems: Course Program by Layers

A. Intro: Operating Systems «Who-What-Why-When-Where-How»
B. A Short Look Back: «Progettazione dei Sistemi Digitali», «Architettura degli Elaboratori»
C. EndPoint: Programmable Computer (CPU, Memory, I/O)
D. Operating Systems - basics: cenni storici, multiprogramming, time sharing, spooling; classificazione.
E. Tipologie di Sistemi di Elaborazione
F. Processes: PCB, Context Switch, Signals, Scheduler
G. Memory Management: Protection, Partitioning (Segmentation, Pagination, page Table), Layers (Cache, TLB, Swapping, LRU)
H. I/O: Buffering, Caching, Scheduling
I. Security & Protection: Threats, Attacks, Risks. CIA: Confidentiality-Integrity-Availability -> Authentication, Authorization, Accounting
J. File System
K. Process Interactions: Race & Communications
L. Starvation, **Livelock**, **Deadlock**

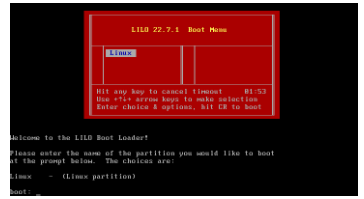
Operating Systems: 3x3 Matrix



	Obiettivi	Funzioni	Servizi
	Astrazione 	File System Sh/GUI/OLTP 	Authentication/ Authorization/ Accounting 
	Virtualizzazione 	Process Mgmt 	IPC System Calls 
	Input/Output 	Memory Mgmt Error Detection, Dual-Mode 	Boot Interrupt Handling 



B	C
Formulas	Formula Results
=1/0	#DIV/0!
=A2/A3	#DIV/0!
=QUOTIENT(A2,A3)	#DIV/0!



Operating Systems: Processi

Processo: definizione

Processo è un programma in esecuzione:

- Indipendente dagli altri programmi
- Che non deve interferire con gli altri programmi
- Che trae giovamento dalla gestione ottimizzata delle risorse
- Che deve sottostare alla allocazione e condivisione ordinata delle risorse rispetto a spazio/tempo

«**Legum servi** sumus ut **liberi** esse possimus»
[M. T. Cicerone]

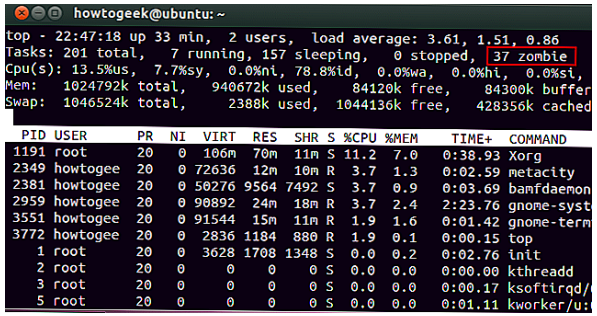
Controllo ferreo sull'accesso alle risorse.

Operating Systems: Processi

Processo != Programma

Processo = Esecuzione delle Istruzioni ➔ Entità attiva

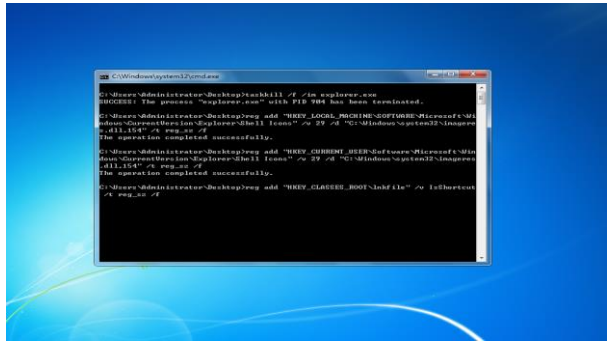
- Contatore di programma
- Valori delle variabili
- Risorse in uso



Programma = Lista di Istruzioni ➔ Entità passiva

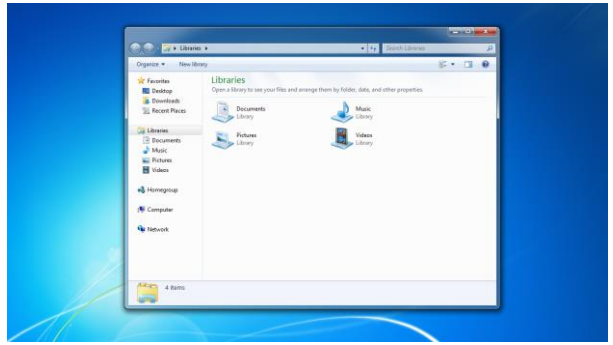
```
def add5(x):
    return x+5

def dotwrite(ast):
    nodename = getNodename()
    label=symbol.sym_name.get(int(ast[0]),ast[0])
    print '    %s [label="%s"' % (nodename, label)
    if isinstance(ast[1], str):
        if ast[1].strip():
            print '= %s"];' % ast[1]
        else:
            print '"]'
    else:
        print '"];'
        children = []
        for n, child in enumerate(ast[1:]):
            children.append(dotwrite(child))
        print '    %s -> {' % nodename,
        for name in children:
            print '%s' % name,
```

due **processi** associati allo stesso programma, sono due differenti istanze di esecuzione dello stesso codice:



0 explorer.exe

1° explorer.exe



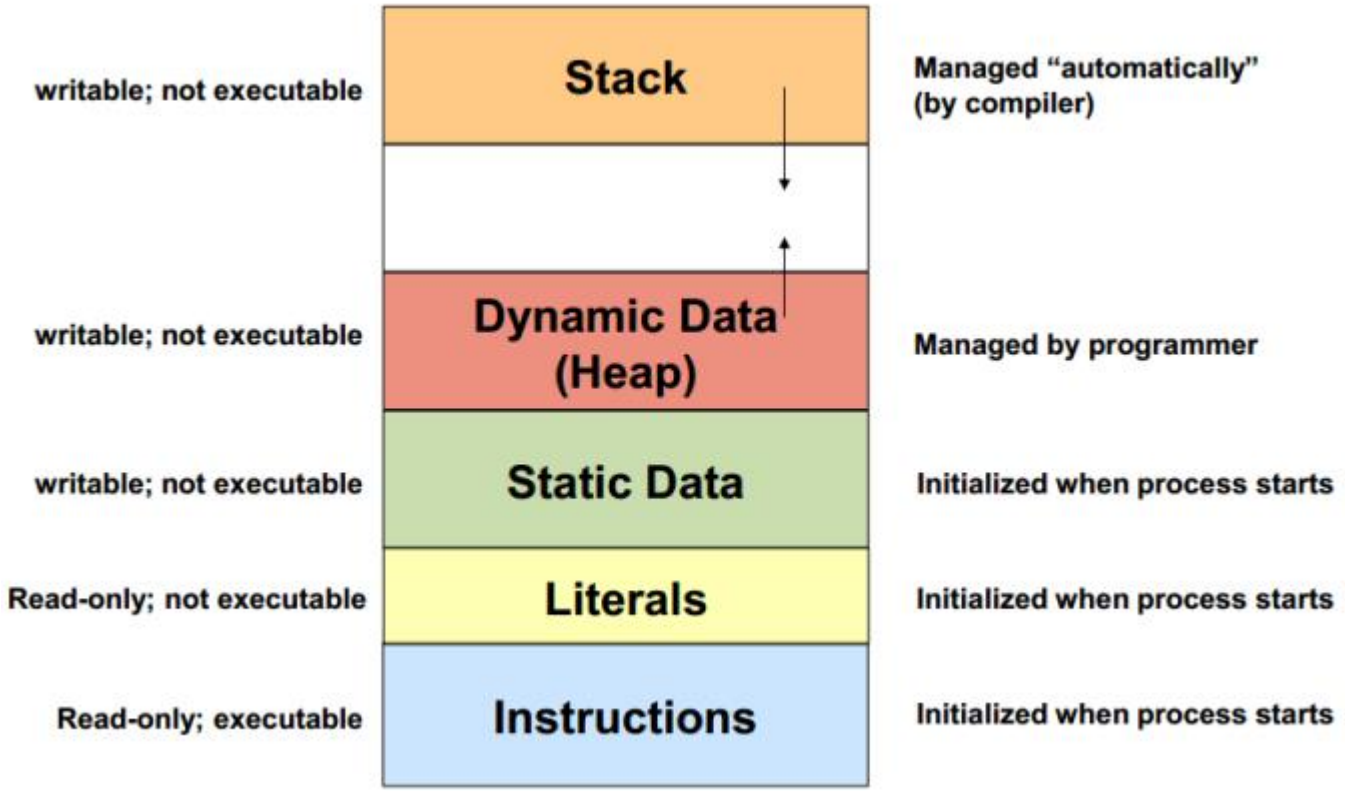
N-simo explorer.exe

Operating Systems: Processi

Memory Layout 1/3

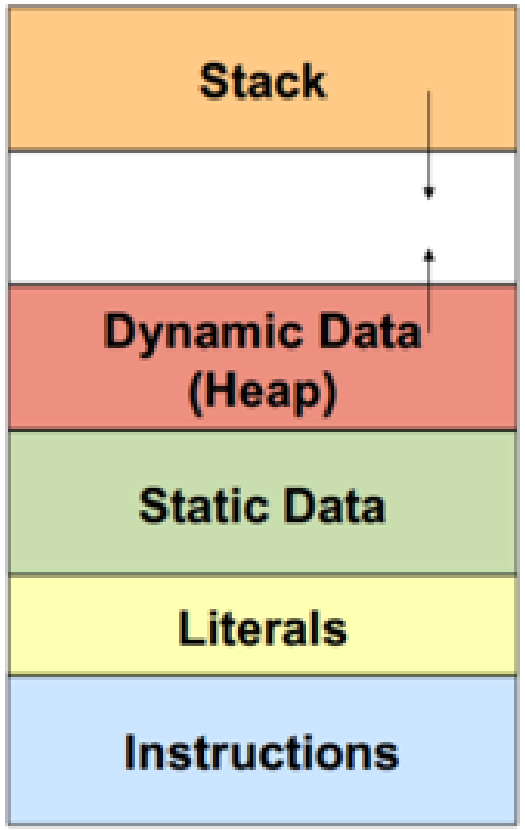
La memoria occupata dal processo è composta di:

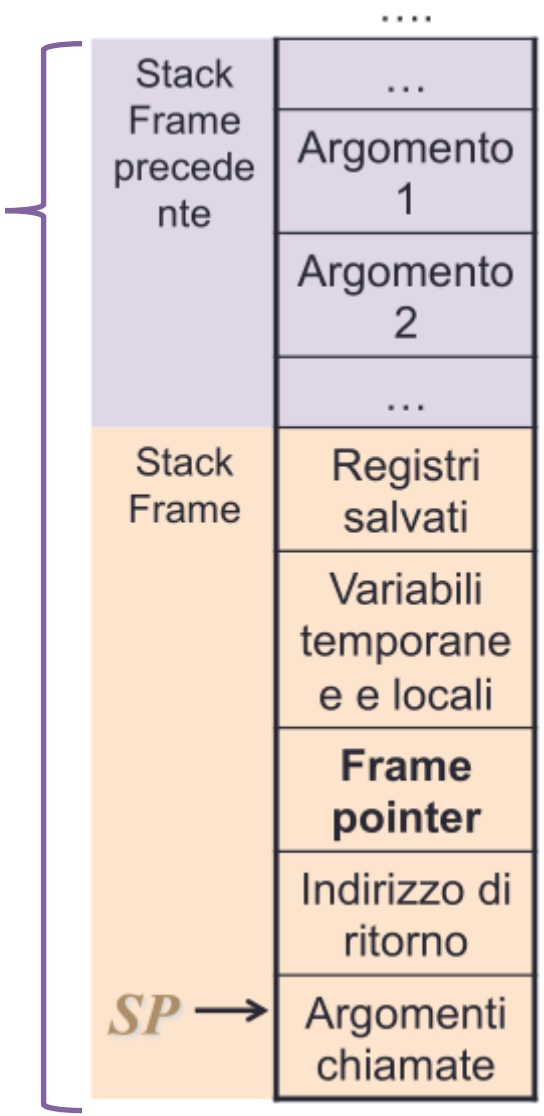
- **Instructions**: codice del programma (puntato dal Program Counter: **PC**)
- **Literals**: parametri del programma
- **Static Data**: variabili globali in memoria centrale (puntato dallo Stack Pointer: **SP**)
- **Heap** (mucchio): dati dinamici
- **Stack** (pila): funzioni (metodi), variabili locali, indirizzi di rientro



Operating Systems: Processi

Memory Layout 2/3





Lo **stack** è una pila di dati che tipicamente cresce verso il basso

- I record di attivazione delle funzioni (stack frame) vengono inseriti (pushed) e rimossi (popped) dallo stack
- Lo stack frame contiene dati relativi ad una funzione:
  - Parametri
  - Variabili locali
  - Dati necessari per ripristinare il frame precedente
- Viene puntato dal Frame Pointer (**FP**)

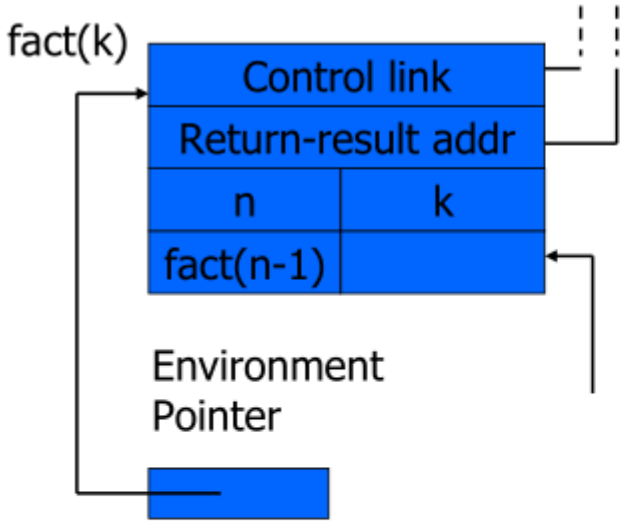
Lo **heap** è una pila di dati che tipicamente cresce verso l'altro

- Il programmatore si occupa di come utilizzare lo heap (non il SO)
- Si deve occupare di allocare (malloc C) e deallocare la memoria
- In Java questo è trasparente al programmatore Garbage Collector)

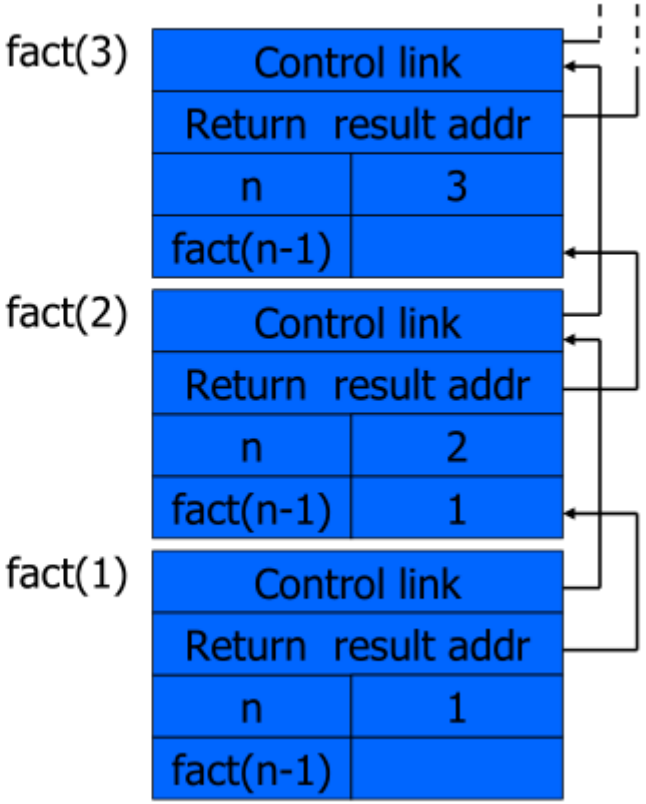
Se **stack** e **heap** collidono si verificano **errori**: Esistono chiamate di sistema per **ridimensionarli**

Operating Systems: Processi

Memory Layout 3/3



fact(n) = if n<= 1 then 1
else n * fact(n-1)



- Classico esempio di esecuzione delle funzione fattoriale, implementata in modo iterativo.
- La stessa funzione viene chiamata tante volte (con parameti sempre diversi) finché non si esce dalla condizione di iterazione.

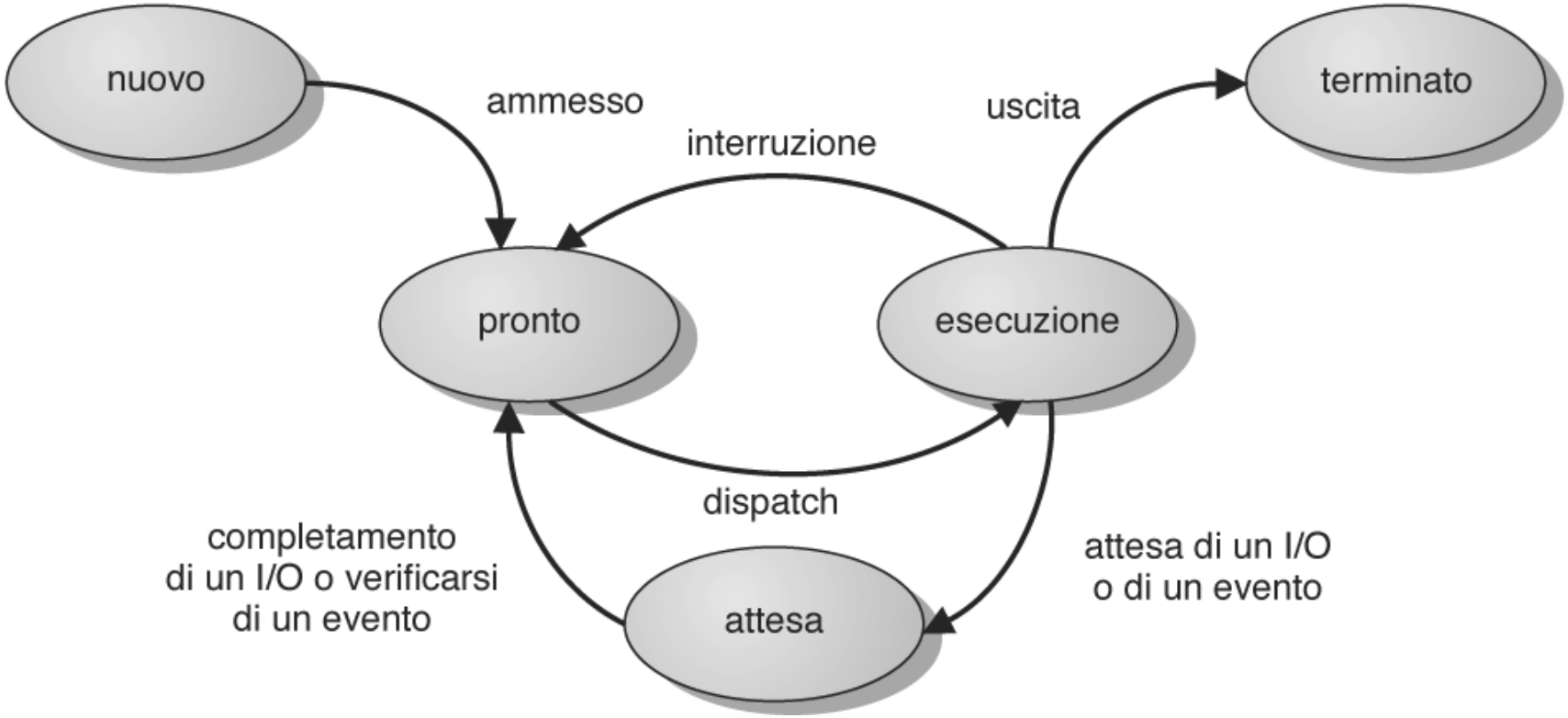
Operating Systems: Processi

Stato

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

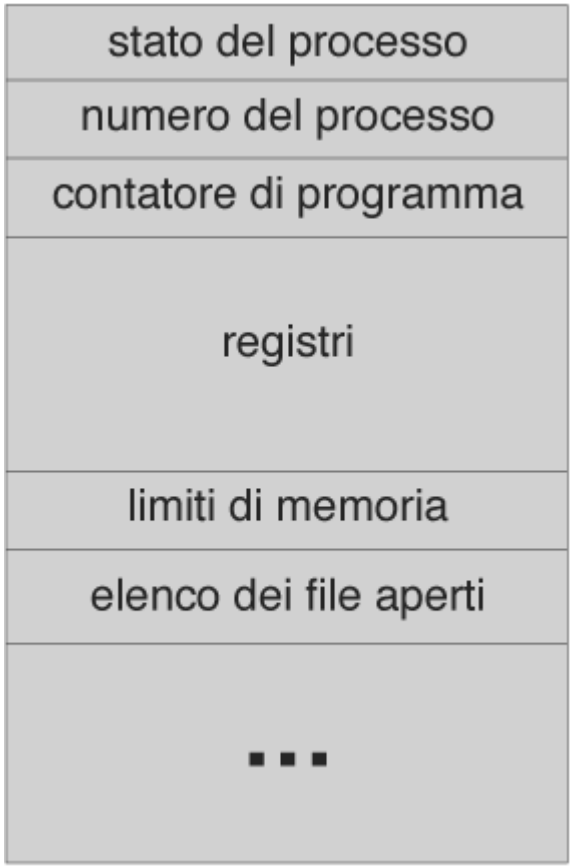


Operating Systems: Processi

PCB

Process Control Block (PCB): Struttura dati del **kernel** che mantiene le informazioni sul processo

- Stato del processo
- Identificatore del processo (Numero)
- Program counter
- i.e. indirizzo istruzione successiva
- Registri della CPU
- Stack Pointer
- Frame Pointer
- I valori devono essere salvati per poter riprendere correttamente l'esecuzione dopo un interruzione
- Informazioni sullo scheduling (es. priorità, etc …)
- Informazioni sulla gestione della memoria (es. tabelle delle pagine, etc …)
- Informazioni di contabilizzazione delle risorse (es. tempo uso CPU, etc …)
- Informazioni sullo stato dell'I/O (es. lista dispositivi assegnati al processo, elenco file aperti, etc …)



Operating Systems: Processi

PCB: esempio Linux

Process Control Block (PCB): Struttura dati del **kernel** che mantiene le informazioni sul processo

```
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;
/* 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;
};
```

Operating Systems: Processi

Scheduling (semplificato): stato pronto ed attesa

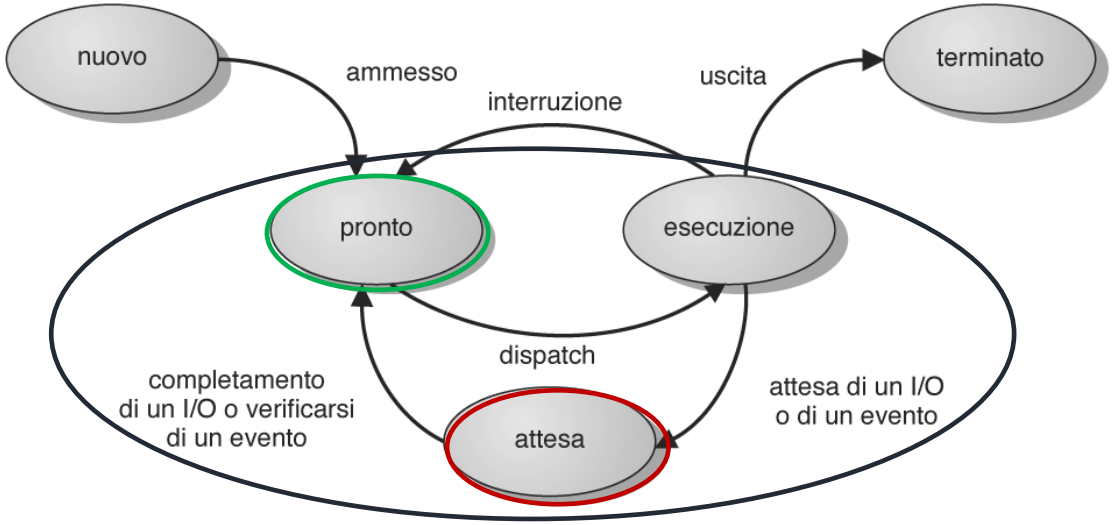
Code di schedulazione dei processi

- 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 (una per ogni dispositivo)

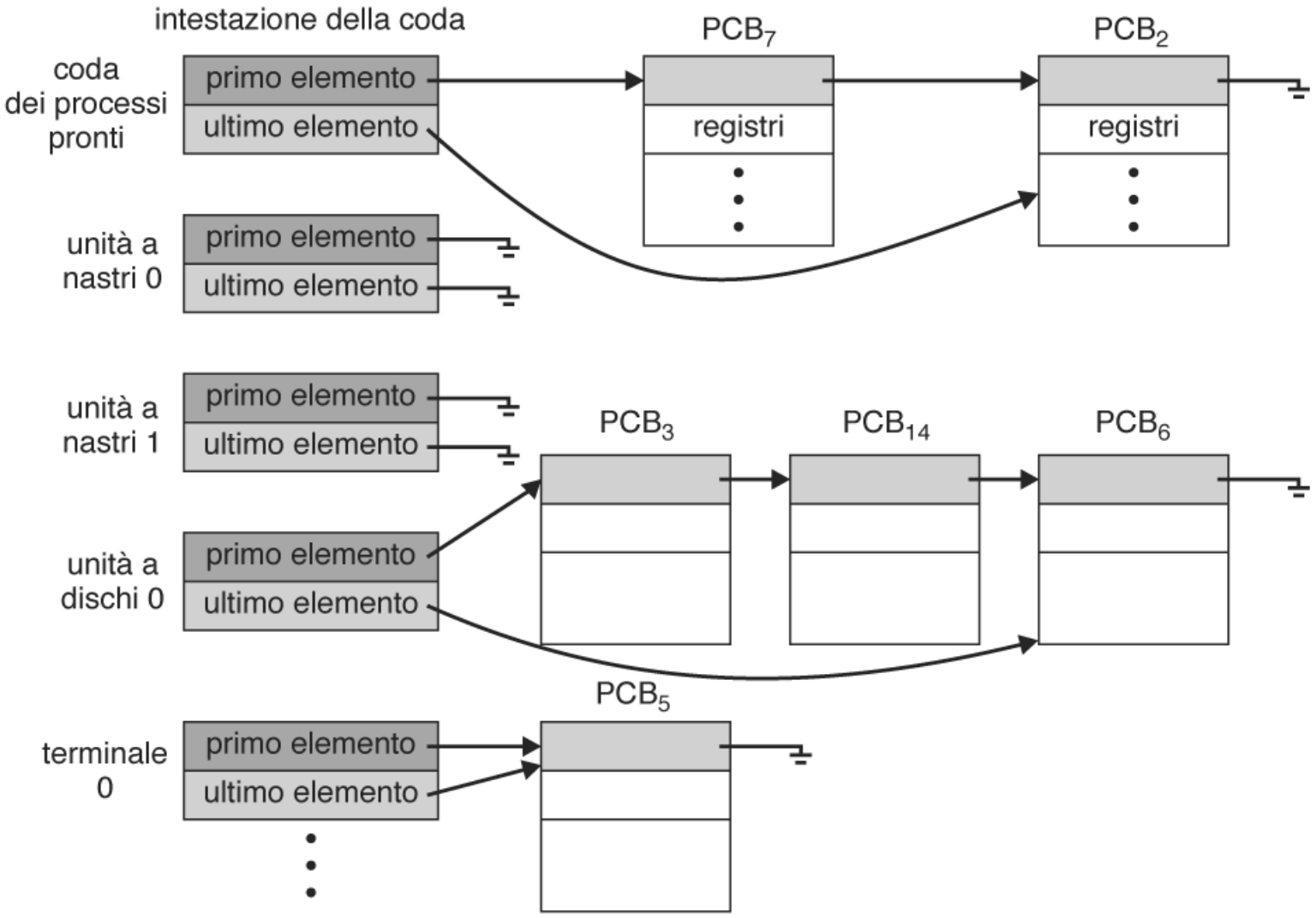
- Il processo si muove fra le varie code



Operating Systems: Processi

Scheduling

Code di schedulazione dei process nei vari stati



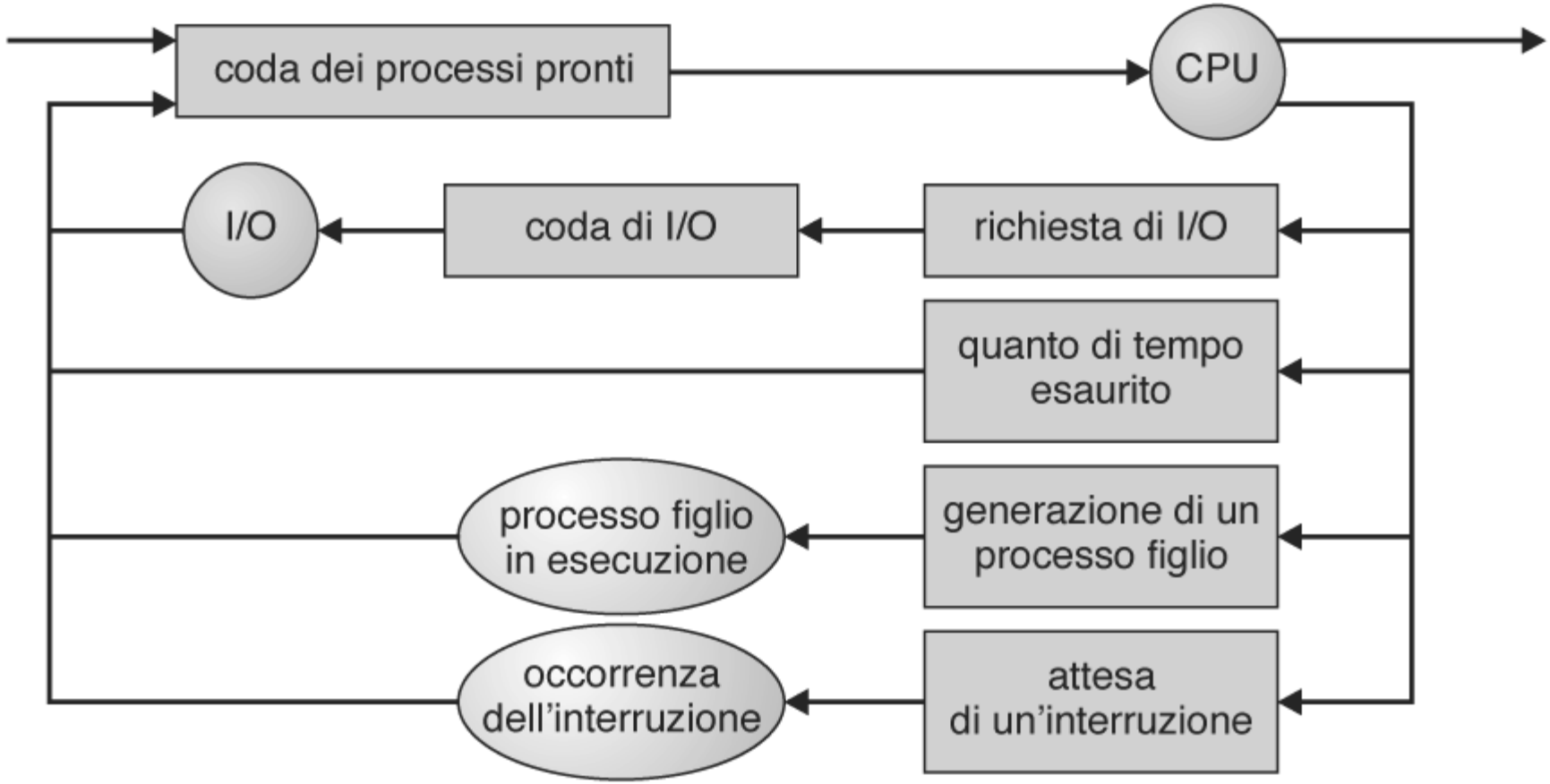
Ogni coda ha due puntatori:

- Primo PCB
- Ultimo PCB

Ogni PCB ha un puntatore al successivo

Operating Systems: Processi

Scheduling: diagramma di accodamento



- Richiesta I/O: il processo torna nella coda pronti al termine dell'I/O
- Interruzione: il processo viene rimesso nella coda dei pronti
- Generazione figlio: il processo torna nella coda dei pronti alla terminazione del figlio
- Quando un processo termina il PCB e le sue risorse sono deallocate

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)

Bilanciamento **I/O-bound** – **CPU-bound** ➔ Buone prestazioni

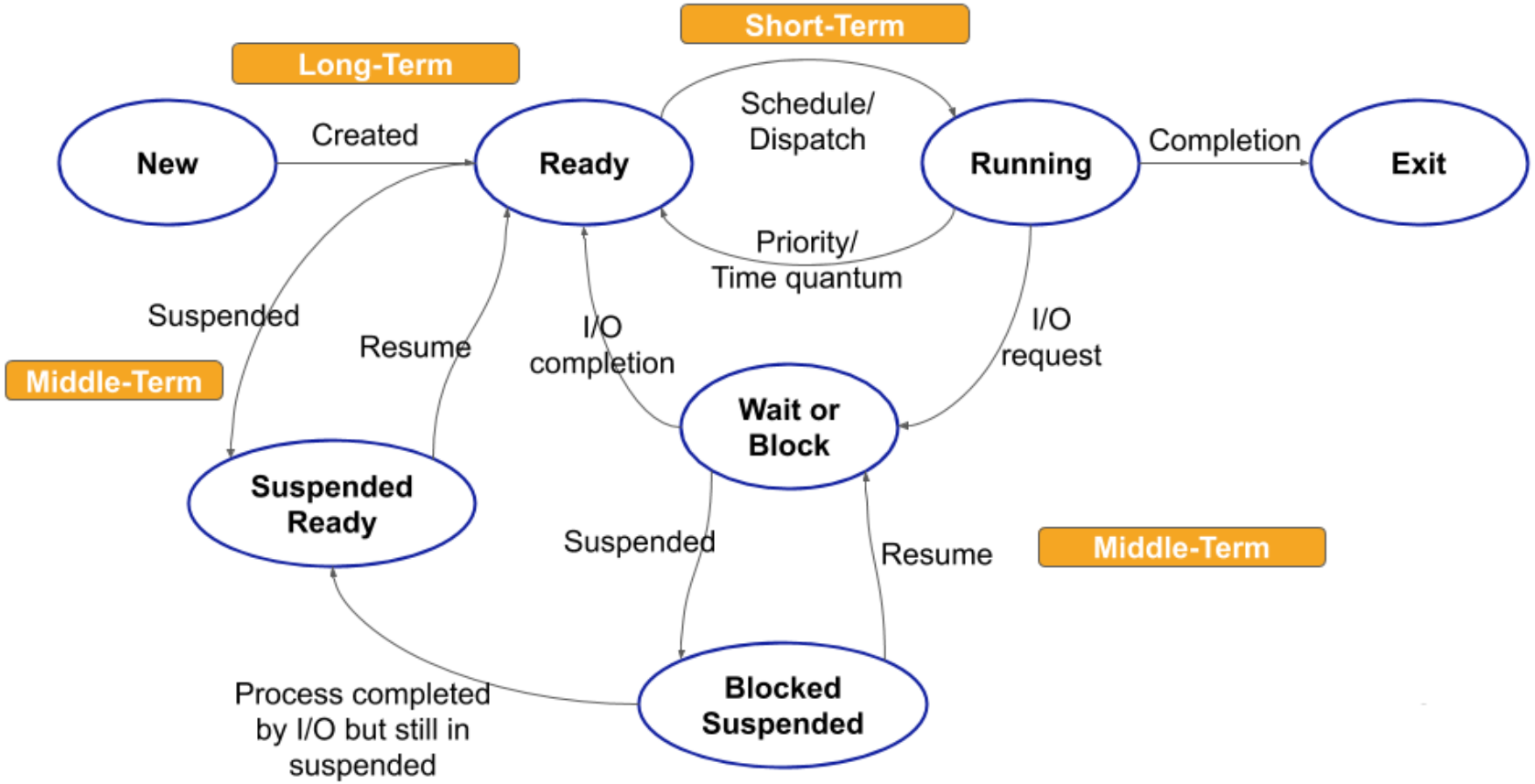
Sbilanciamento **I/O-bound** – **CPU-bound** ➔ Coda "Pronto" o coda "Attesa" vuote

Operating Systems: Processi

Schedulatori

- **Short-Term** (Schedulatore a breve termine): seleziona quale processo (in memoria) deve essere eseguito e alloca la CPU ad esso ogni ms (jiffies)

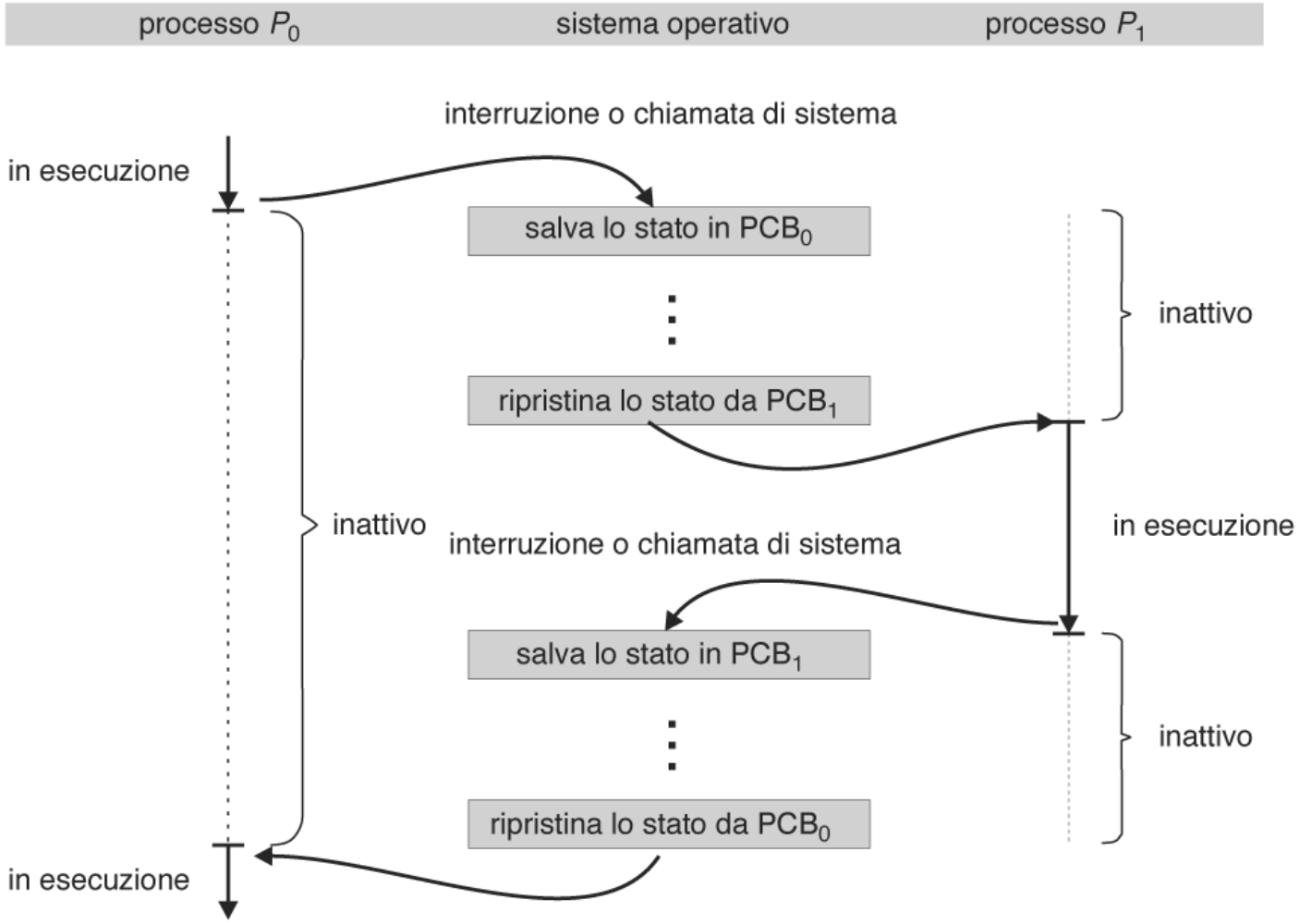
- **Middle-Term** (Schedulatore a medio termine): esegue lo swapping
  - Rimuove un processo dalla memoria centrale e lo pone in memoria di massa
  - Supportato solo da alcuni SO come i sistemi time-sharing



- **Long-Term** (Schedulatore a lungo termine): seleziona quale processo (attualmente in memoria di massa) deve essere inserito nella coda dei processi pronti, con frequenza nell'ordine dei secondi/minuti:
  - Controlla il grado di multi-programmazione
  - Stabile se velocità di creazione processi = velocita terminazione processi
  - Richiamato solo quando un processo abbandona il sistema (termina)
  - Assente nei sistemi UNIX e Windows

Operating Systems: Processi

Context Switch



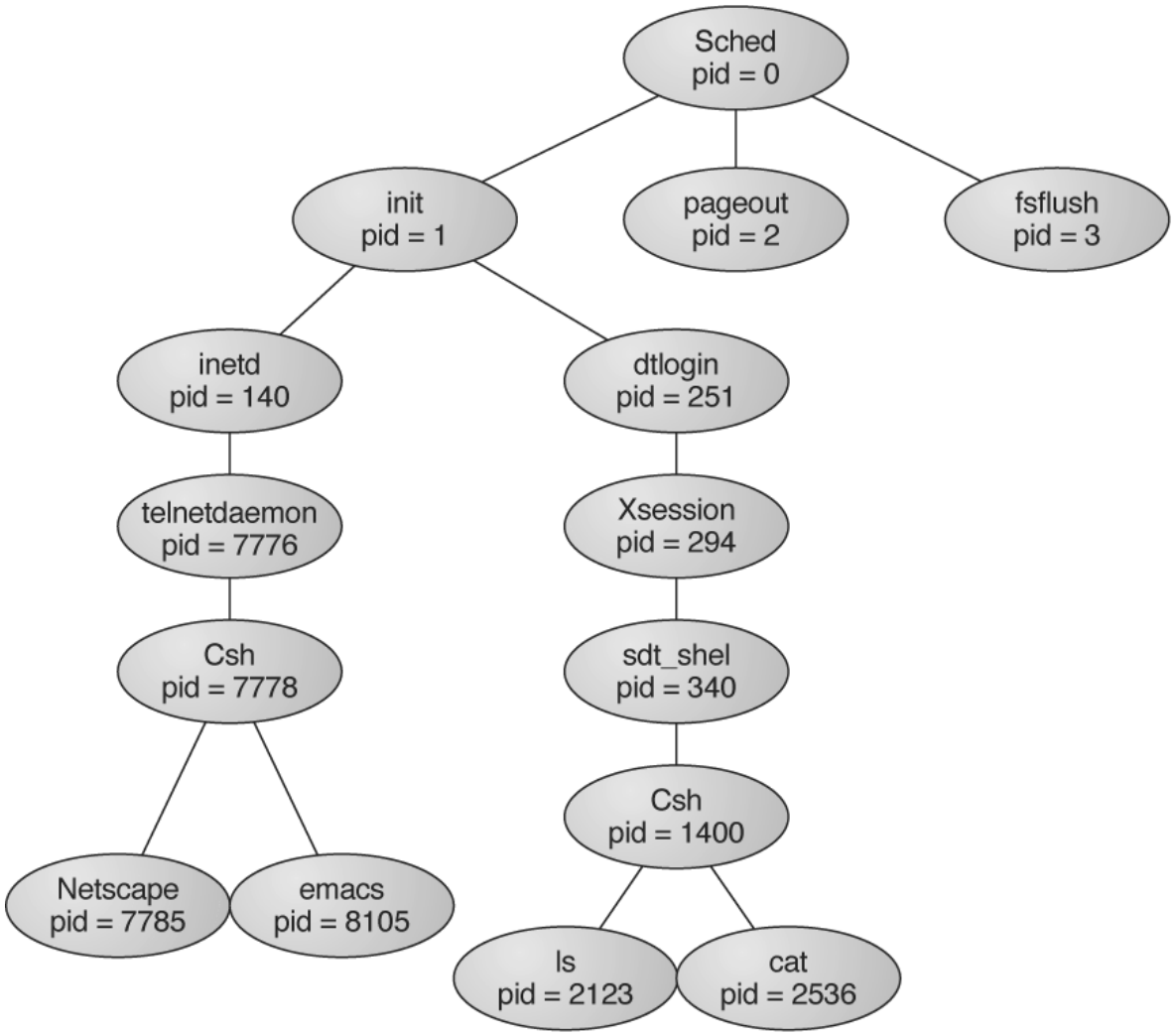
Context Switch (Cambio di Contesto): passaggio della CPU da un processo ad un altro

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

- Il tempo per il cambio di contesto è puro tempo di gestione del sistema (non vengono compiute operazioni utili per la computazione dei processi)
- I tempi (10ms) per i cambi di contesto dipendono dal supporto hardware (e.g. registri disponibili e/o registri dedicati). <

Operating Systems: Processi

Nuovo Processo: Creazione (Generazione) 1/2



Un processo in esecuzione può creare numerosi sottoprocessi usando un'apposita chiamata di sistema (create_process):

- Processo generante = processo padre
- Processo generato = processo figlio

Ogni processo ha un identificatore univoco PID (intero)

In unix si usa la chiamata fork(): nuovo processo

➔ albero di processi (visibile ad es. tramite ps -el)

➔ Inizialmente, il figlio è la copia del padre (es. codice binario, spazio degli indirizzi

➔ Ritorna un valore

- ➔ 0: figlio
- ➔ <>0: padre
- ➔ Exec() sovrascrive lo spazio degli indirizzi del chiamante caricando un nuovo programma

Esecuzione del processo padre dopo la creazione del figlio:

- Continua in modo concorrente, oppure
- Attende la terminazione del figlio ➔ wait()

Operating Systems: Processi

Nuovo Processo: Creazione (Generazione) 2/2

- La chiamata fork() crea un nuovo processo. Il valore di ritorno vale
  - 0 per il figlio ➔ 1° else
  - Diverso da 0 per il padre ➔ 2° else

La fork crea una copia dello spazio degli indirizzi del genitore
Entrambi, padre e figlio, continuano l'esecuzione dalla chiamata successiva alla fork

La chiamata wait(NULL) blocca il chiamante fino alla terminazione di un figlio (il primo che termina)

waitpid(…, int pid, …) permette di attendere la terminazione di uno specific figlio

```
int main()
{
  pid_t pid;

  // fork a child process
  pid = fork();

  if (pid < 0) { // error occurred
      fprintf(stderr, "Fork Failed\n");
      exit(-1);
  }
  else if (pid == 0) { // child process
      printf("I am the child %d\n",pid);
      execlp("/bin/ls","ls",NULL);
  }
  else { // parent process
    // parent will wait for the child to
    complete
      printf("I am the parent %d\n",pid);
      wait(NULL);

      printf("Child Complete\n");
      exit(0);
  }
}
```

Operating Systems: Processi

Terminazione del Processo: Normale ed Anomala

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
  - File aperti
  - Memoria fisica e virtuale
  - Aree di memoria per I/O

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
- In unix la terminazione a cascata non esiste: I processi figli vengono "adottati" dal processo init

Mancata terminazione tramite la chiamata wait(), porta ad un processo "zombie": la cui elaborazine è terminate ma cui il padre non ha dato il consenso per la terminazione.