SISTEMI OPERATIVI

(MODULO DI INFORMATICA II)

I processi

Prof. Luca Gherardi

Prof.ssa Patrizia Scandurra (anni precedenti)

Università degli Studi di Bergamo

a.a. 2012-13

Sommario

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

Il concetto di processo (1)

- Inizialmente i sistemi di calcolo eseguivano un programma alla volta
  - Completo controllo del sistema
  - e dell'accesso alle risorse

- Oggi più programmi vengono eseguiti contemporaneamente
  - Condivisione risorse
  - Concorrenza
- È necessario un controllo più ferreo sull'accesso alle risorse

- Da questa esigenza nasce il concetto **di processo (di elaborazione)**
- Sistema = insieme di processi

Il concetto di processo (2)

- Cosa è un processo?

- Un SO esegue una varietà di attività:
  - Sistema a lotti (batch system) – lavoro (jobs)
  - Sistema a condivisione di tempo (time-sharing) – programmi utente (tasks)

- Oltre ai jobs e task ci sono le attività interne del SO

- Un sistema esegue sempre un numero elevato di attività (basti vede il task manager [win] o top [unix])

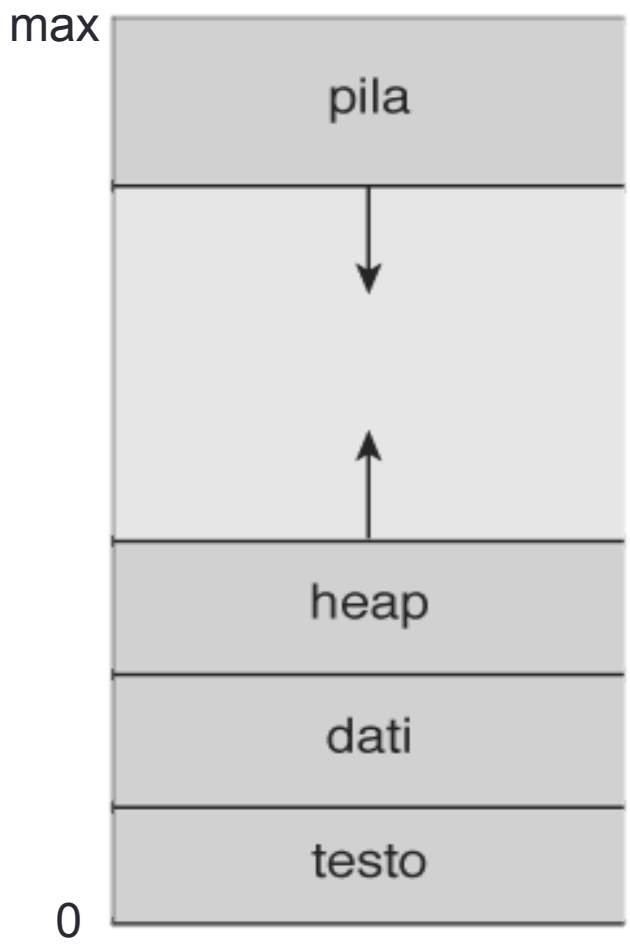
Il concetto di processo (2)

- Queste attività sono dette **processi**
- I termini **job** e **processo** spesso usati come sinonimi
- **Processo** – "un programma in esecuzione"
  - l'esecuzione del processo deve progredire in modo sequenziale
- È solo una prima approssimazione,
- Un processo è molto di più ….

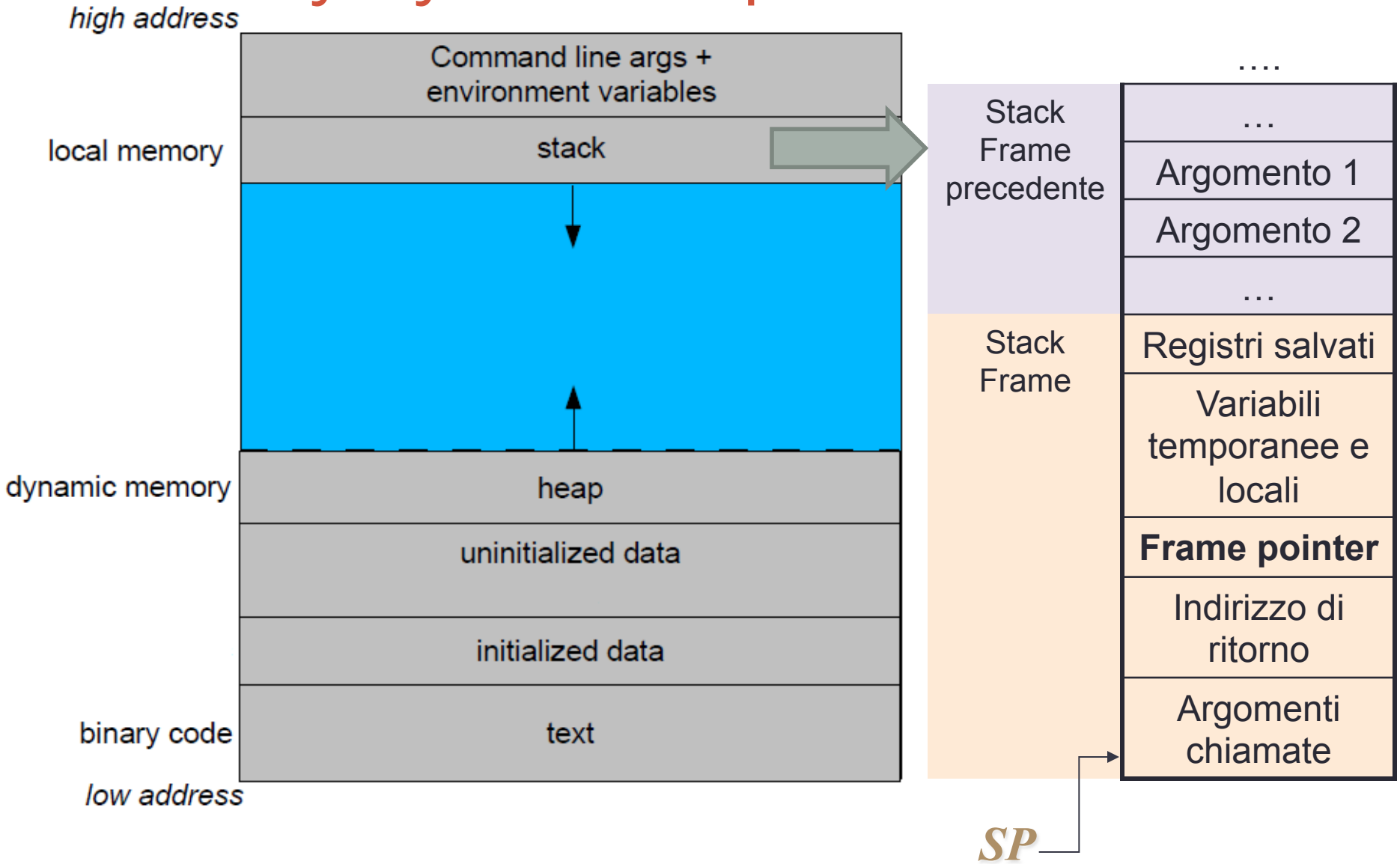
Il concetto di processo (3)

In realtà un processo è composto da:

- **Codice** del programma (text o code section)
- **Dati** del programma
  - Variabili globali in memoria centrale (dati)
  - Variabili temporanee (stack o pila)
    - Parametri metodi, variabili locali, indirizzi di rientro
  - Variabili allocate dinamicamente (heap)
    - Allocazione e deallocazione della memoria all'interno di questo segmento sono a carico del programmatore
  - Attività corrente (non in memoria centrale)
    - Registri del processore
    - Contatore di programma (indirizzo istruzione successiva)

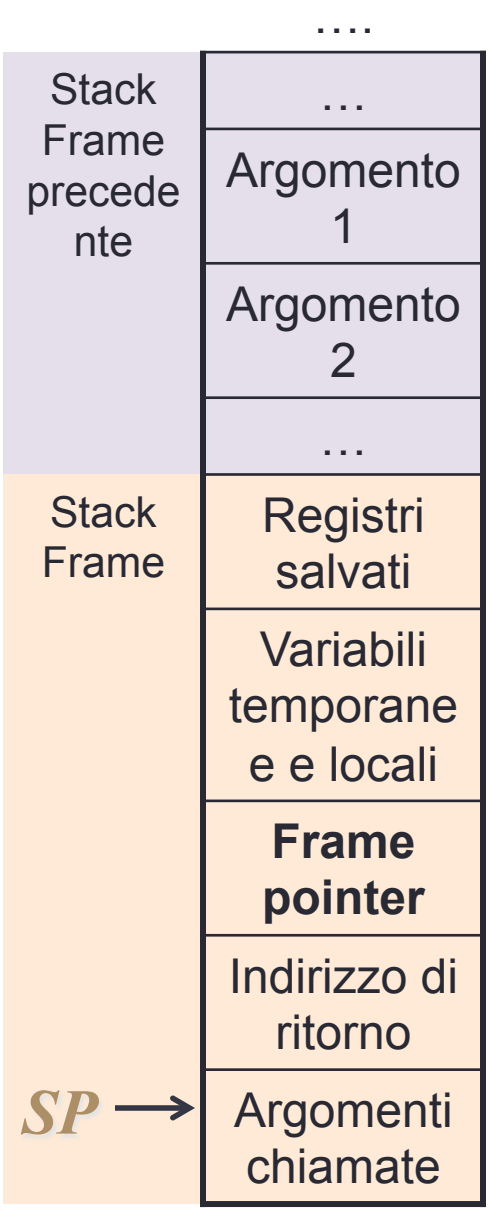


Memory layout di un processo

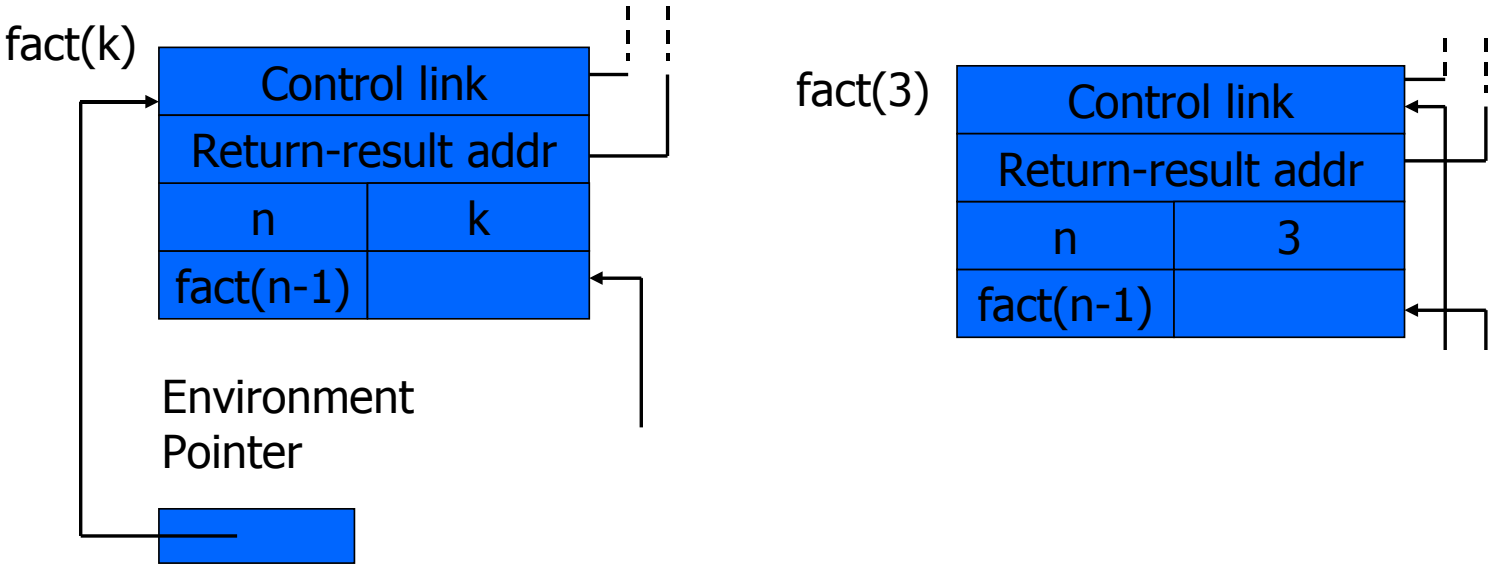


Memory layout di un processo

- 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 rismossi (popped) dallo stack
- Lo stack frame contiene dati relativi ad una funzione
  - Parametri
  - Variabili locali
  - Dati necessari per ripristinare il frame precedente

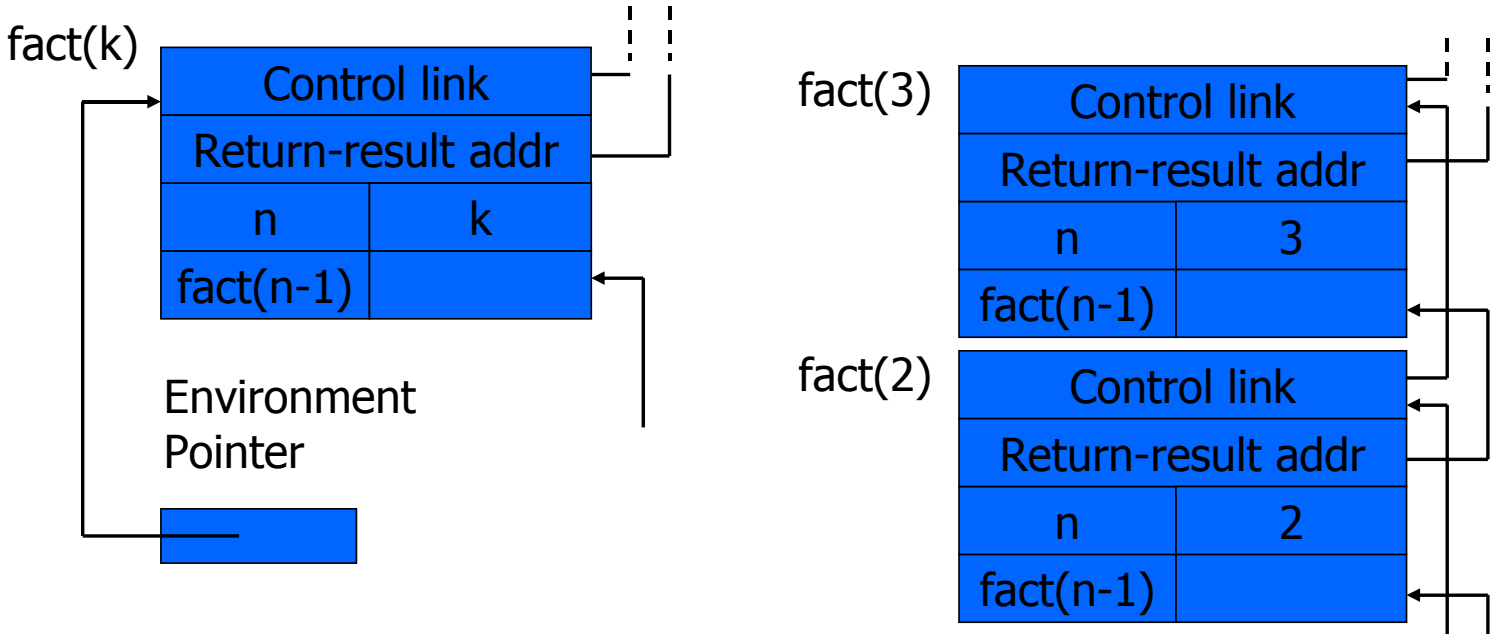


Esempio di uso dello stack



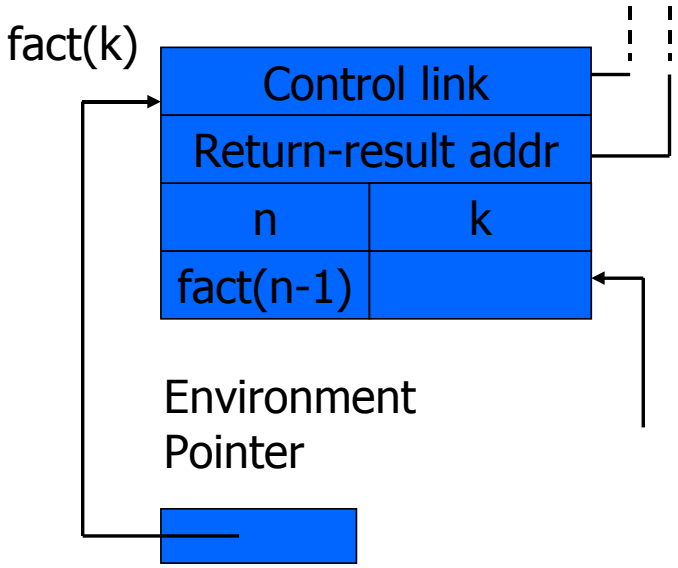
fact(n) = if n<= 1 then 1
else n * fact(n-1)

Esempio di uso dello stack

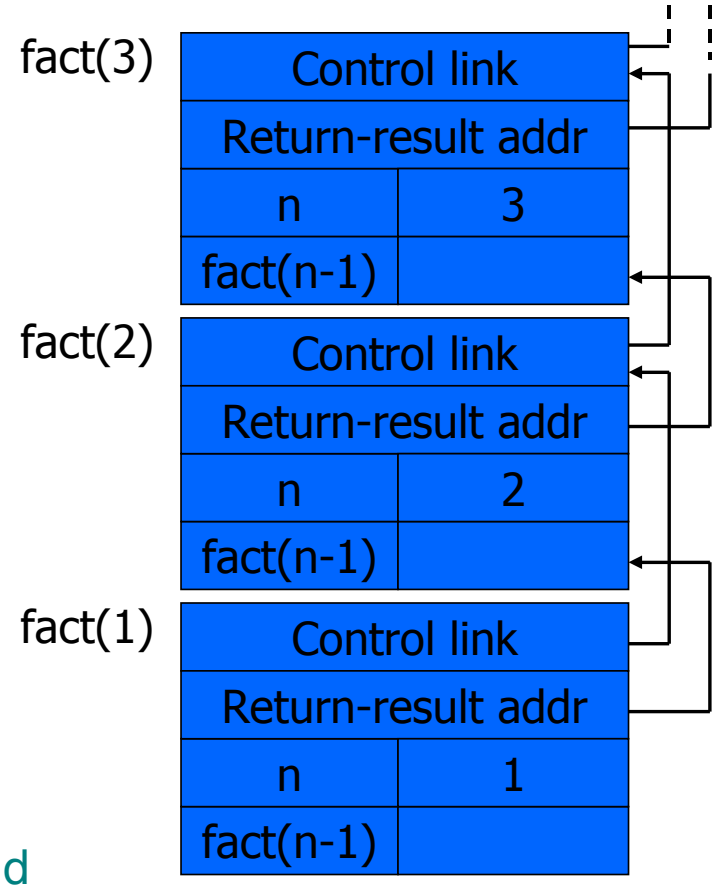


fact(n) = if n<= 1 then 1
else n * fact(n-1)

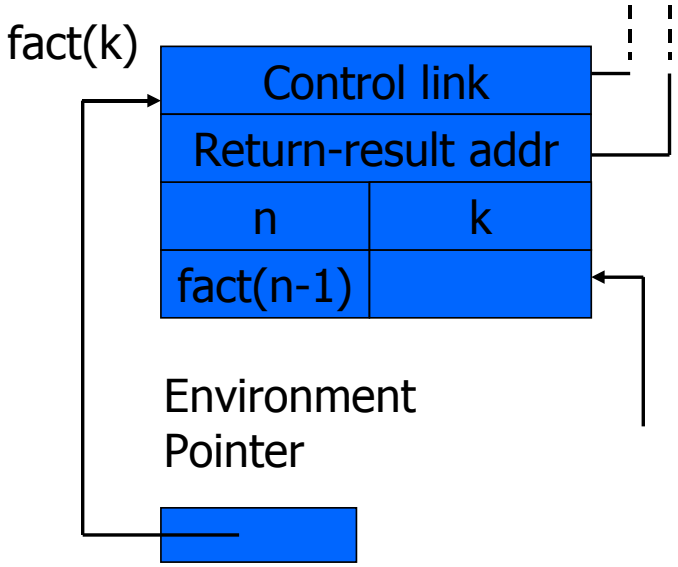
Esempio di uso dello stack



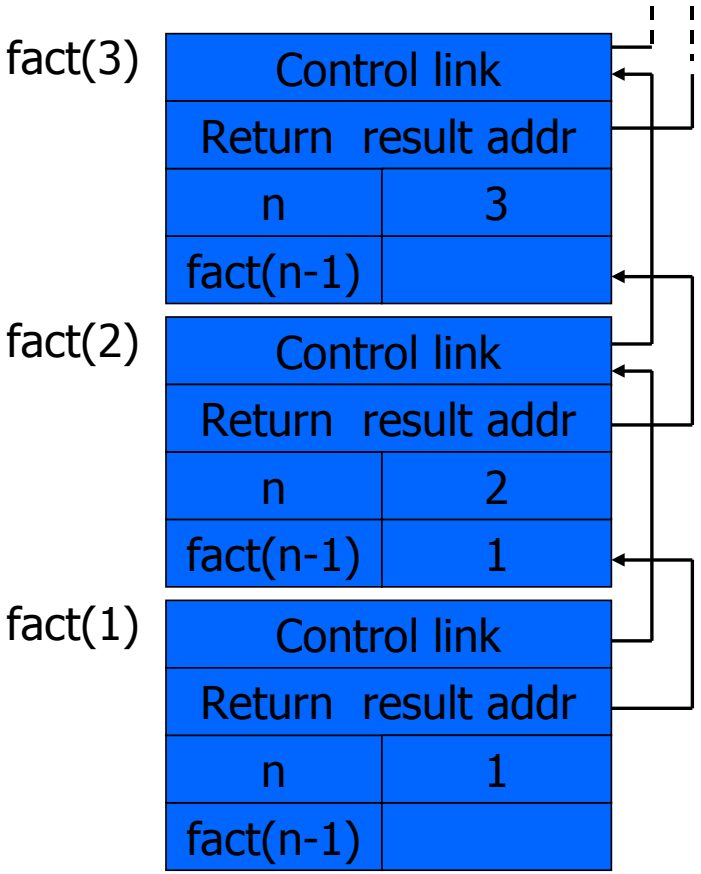
fact(n) = if n<= 1 then 1
else n * fact(n-1)



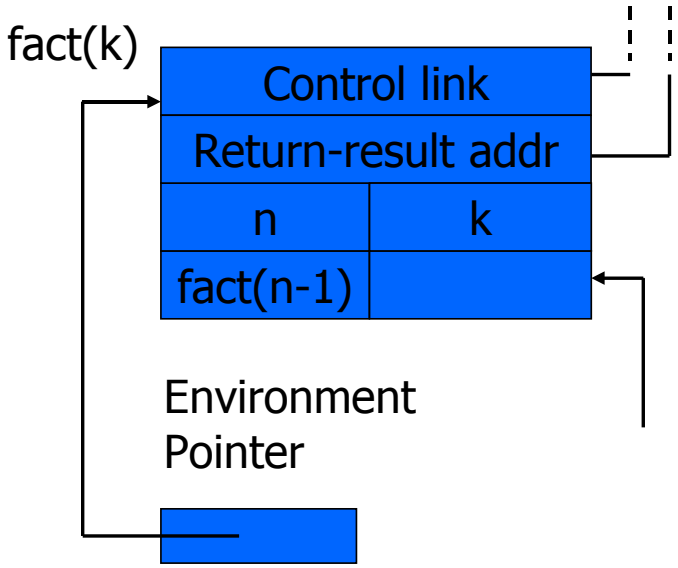
Esempio di uso dello stack

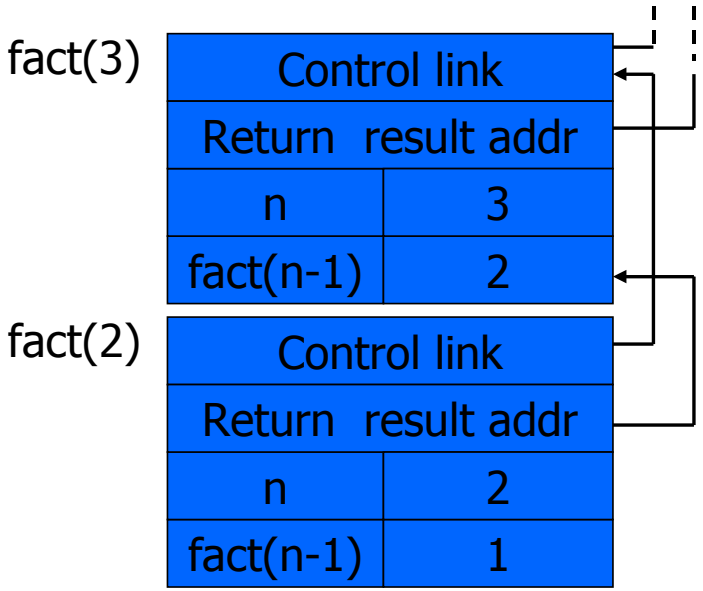


fact(n) = if n<= 1 then 1
else n * fact(n-1)



Esempio di uso dello stack

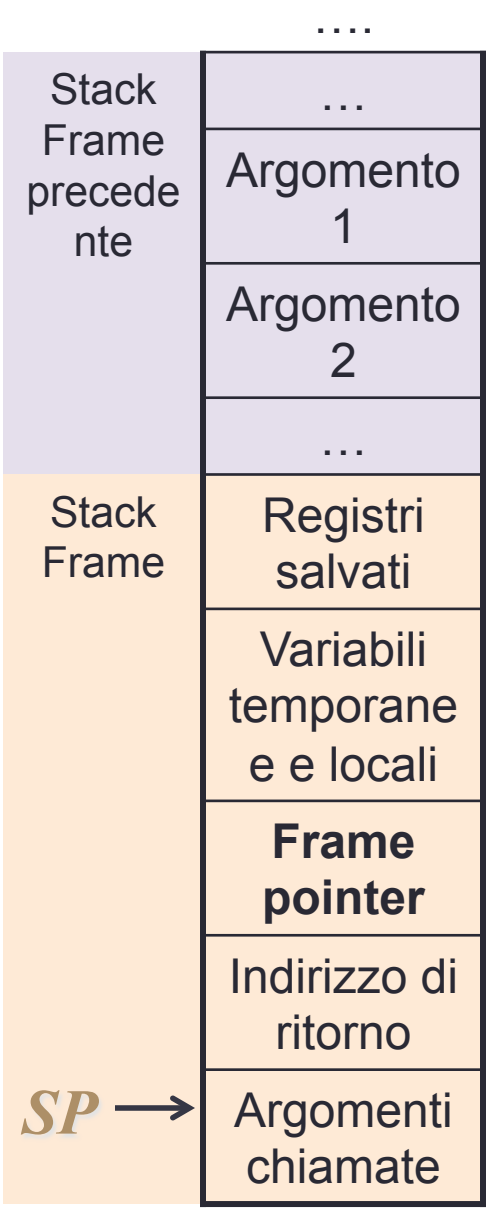




fact(n) = if n<= 1 then 1
else n * fact(n-1)

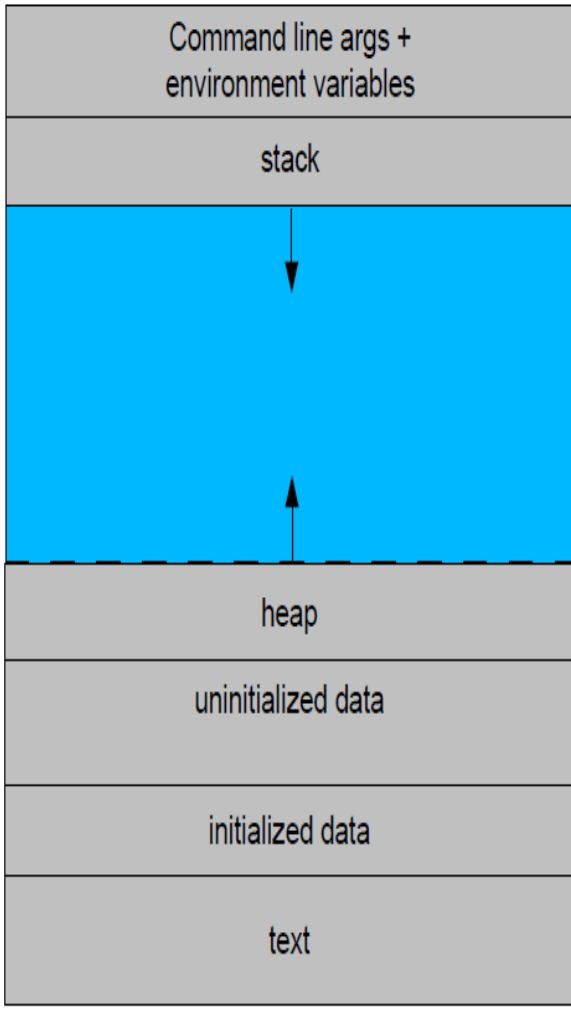
Memory layout di un processo

- La CPU usa un registro chiamato **Stack Pointer (SP)**
  - Punta al top dello stack
  - Li vengono fati i push e i pop
- Esiste un secondo registro detto **Frame Pointer (FP)**
  - Punta ad una cella fissa nel frame
  - Riferimento ad altre celle specificando offset da FP
  - Sarebbe scomodo farlo con SP (cambia)



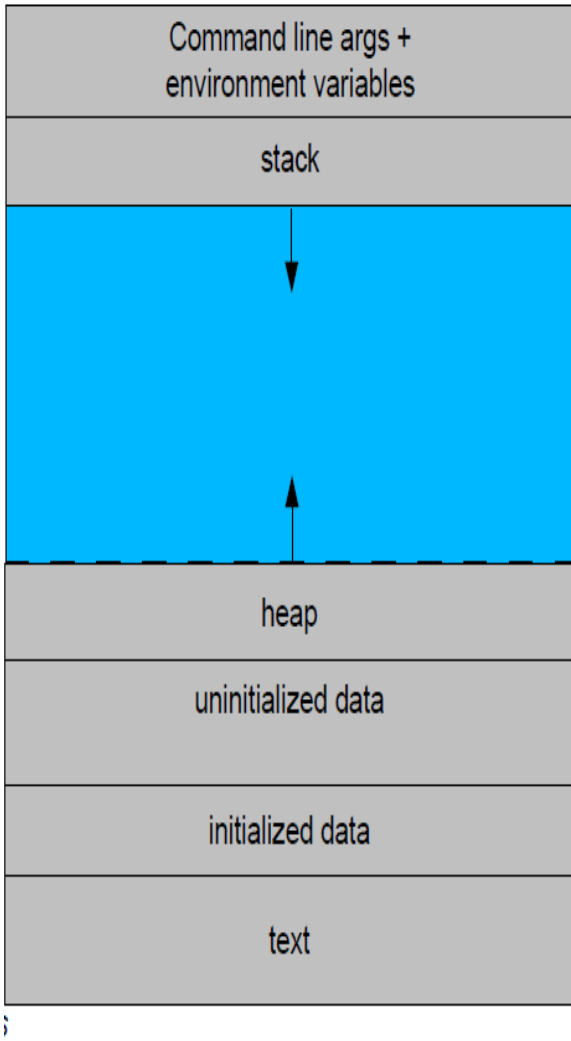
Memory layout di un processo

- 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



Memory layout di un processo

- La sezione **dati** memorizza variabili globali (in C variabili statiche)
  - Inizializzati: permesso di accesso in lettura, scrittura ed esecuzione
  - Non inizializzati: lettura ed esecuzione
- **Text** contiene le istruzioni in linguaggio macchina
  - È puntato dal **Program Counter**



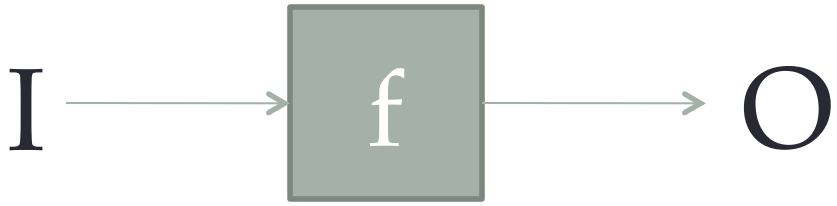
Programma ≠ Processo

- **Programma**
  - Entità passiva
  - Lista istruzioni
- **Processo**
  - Entità attiva
  - Contatore di programma
  - 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!

Evoluzione della computazione di un processo

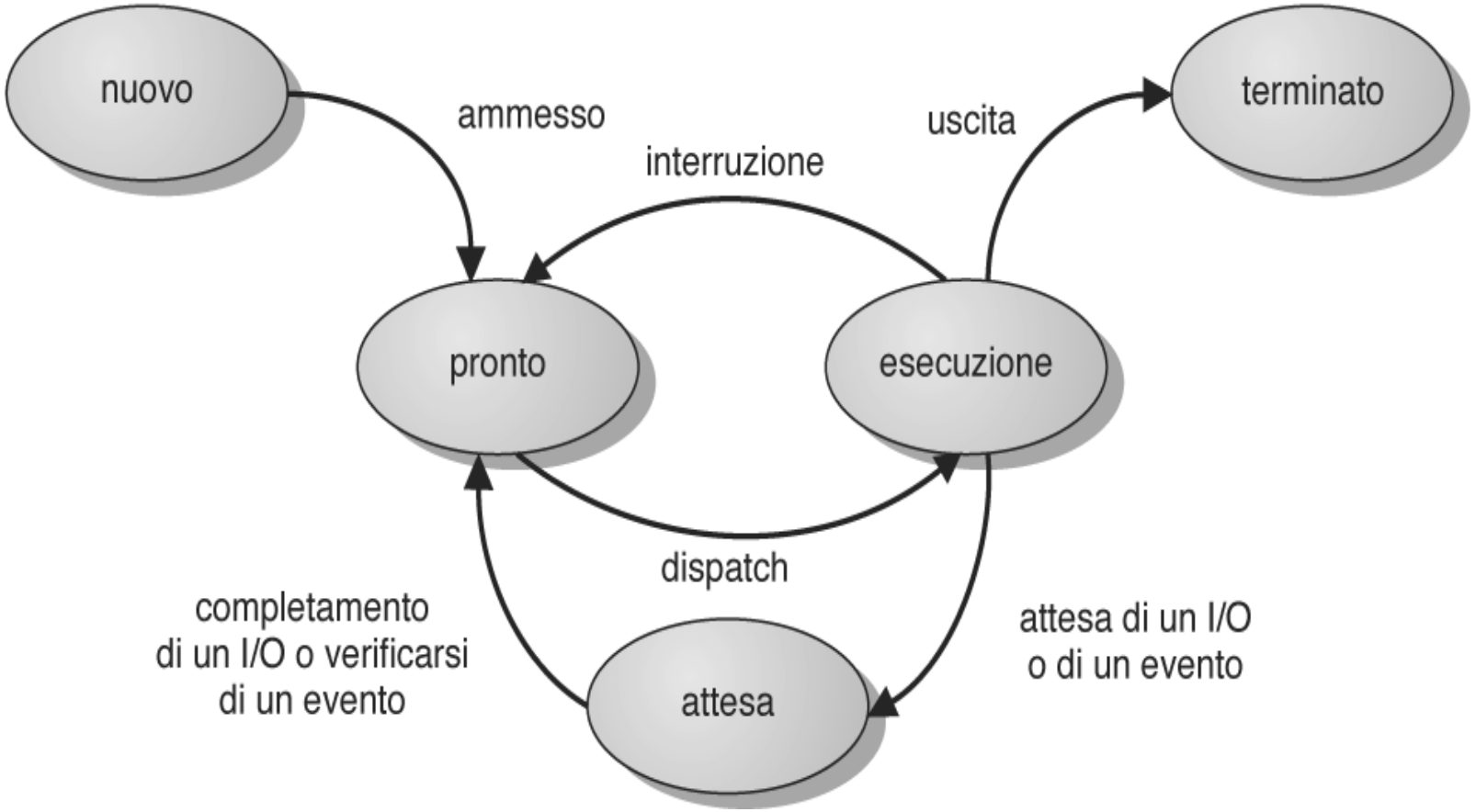
- Il processo è una **funzione** che trasforma informazioni eseguendo le istruzioni del programma
  - partendo dai valori iniziali
    - eventualmente acquisiti durante l'esecuzione stessa attraverso le periferiche
  - e producendo i risultati finali
    - emessi attraverso le periferiche



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
- **1 solo** processo **per unità** di elaborazione può essere in esecuzione

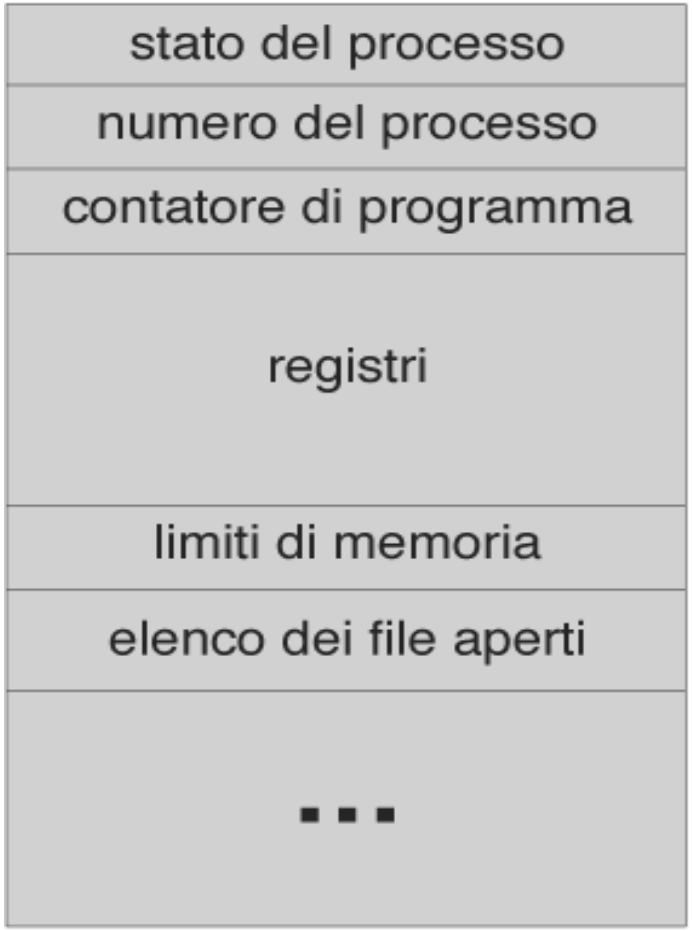
Diagramma degli stati di un processo



Supporti per la gestione dei processi (1)

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

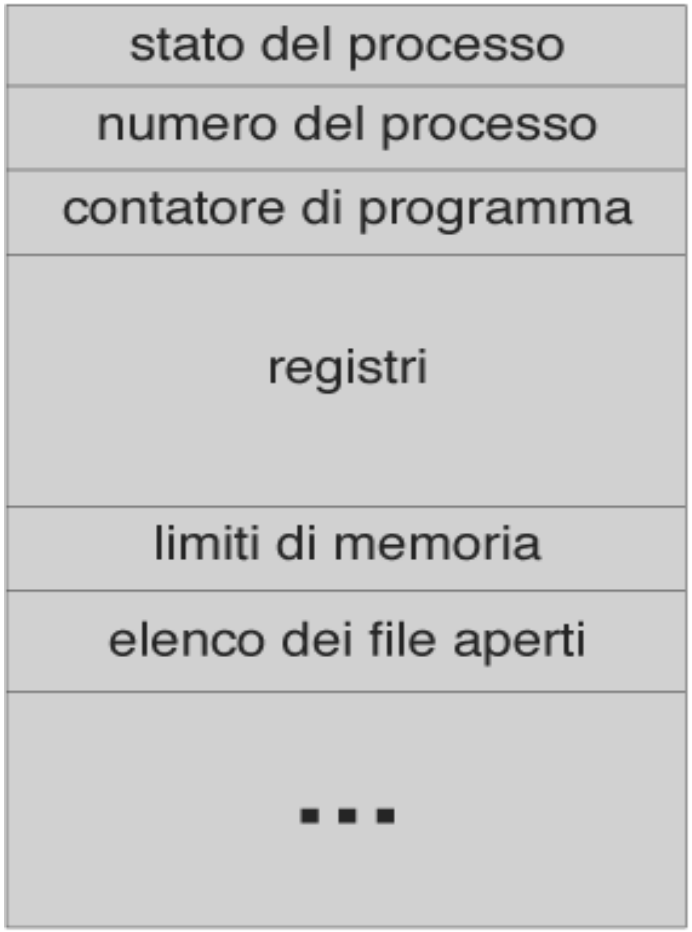
- Stato del processo
- Identificatore del processo (Numero)
- Program counter
  - i.e. indirizzo istruzione successiva
- Registri della CPU
  - Stack Pointer
  - Frame Pointer
  - I valori devono essere salvati per poter riprendere correttamente l'esecuzione dopo un interruzione



Supporti per la gestione dei processi (1)

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

- Informazioni sullo scheduling
  - E.g. priorità, …
- Informazioni sulla gestione della memoria
  - E.g. tabelle delle pagine, …
- Informazioni di contabilizzazione delle risorse
  - E.g. tempo uso CPU, …
- Informazioni sullo stato dell'I/O
  - E.g. lista dispositivi assegnati al processo, elenco file aperti, …



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

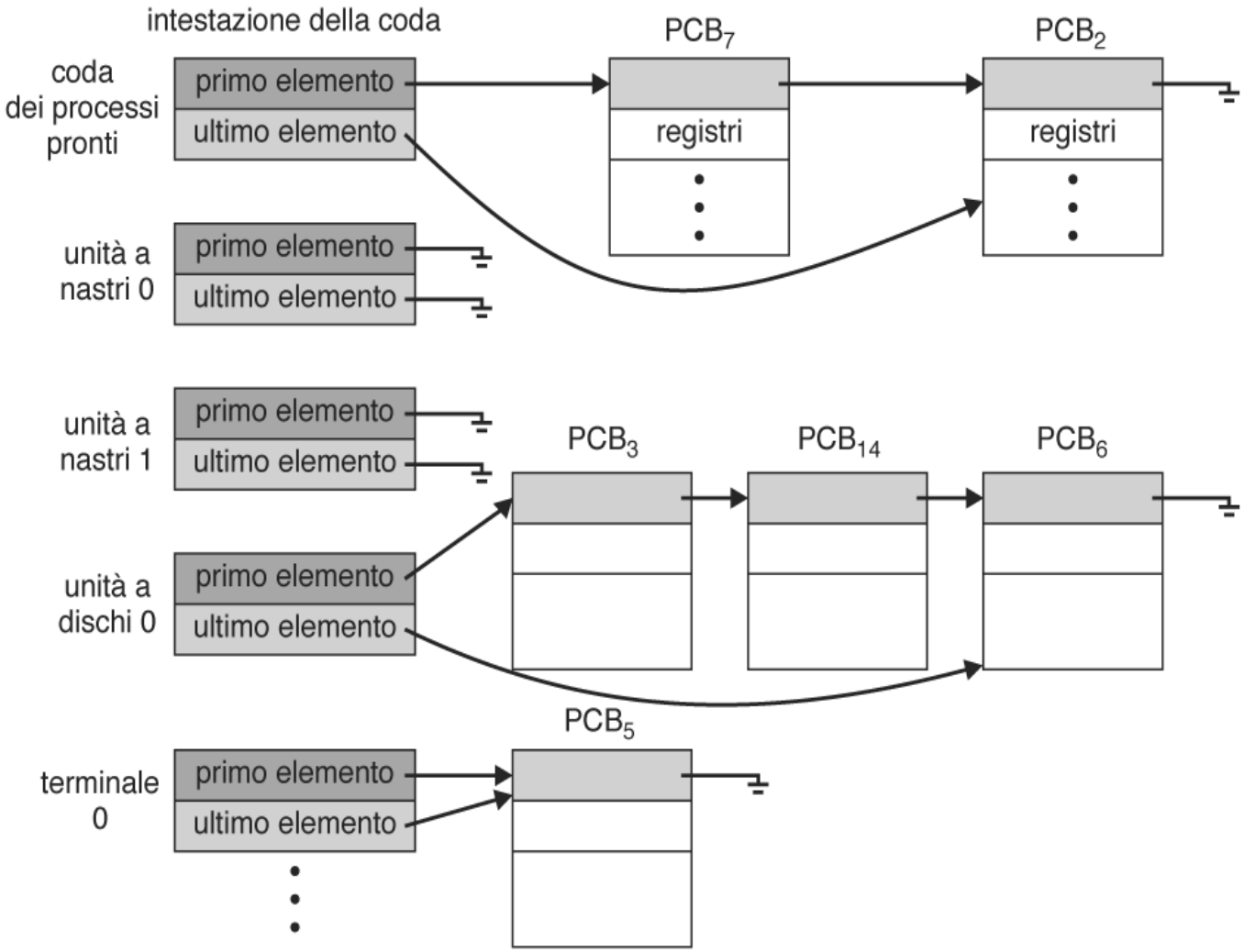
Supporti per la gestione dei processi (2)

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 (una per ogni dispositivo)
- Il processo si muove fra le varie code

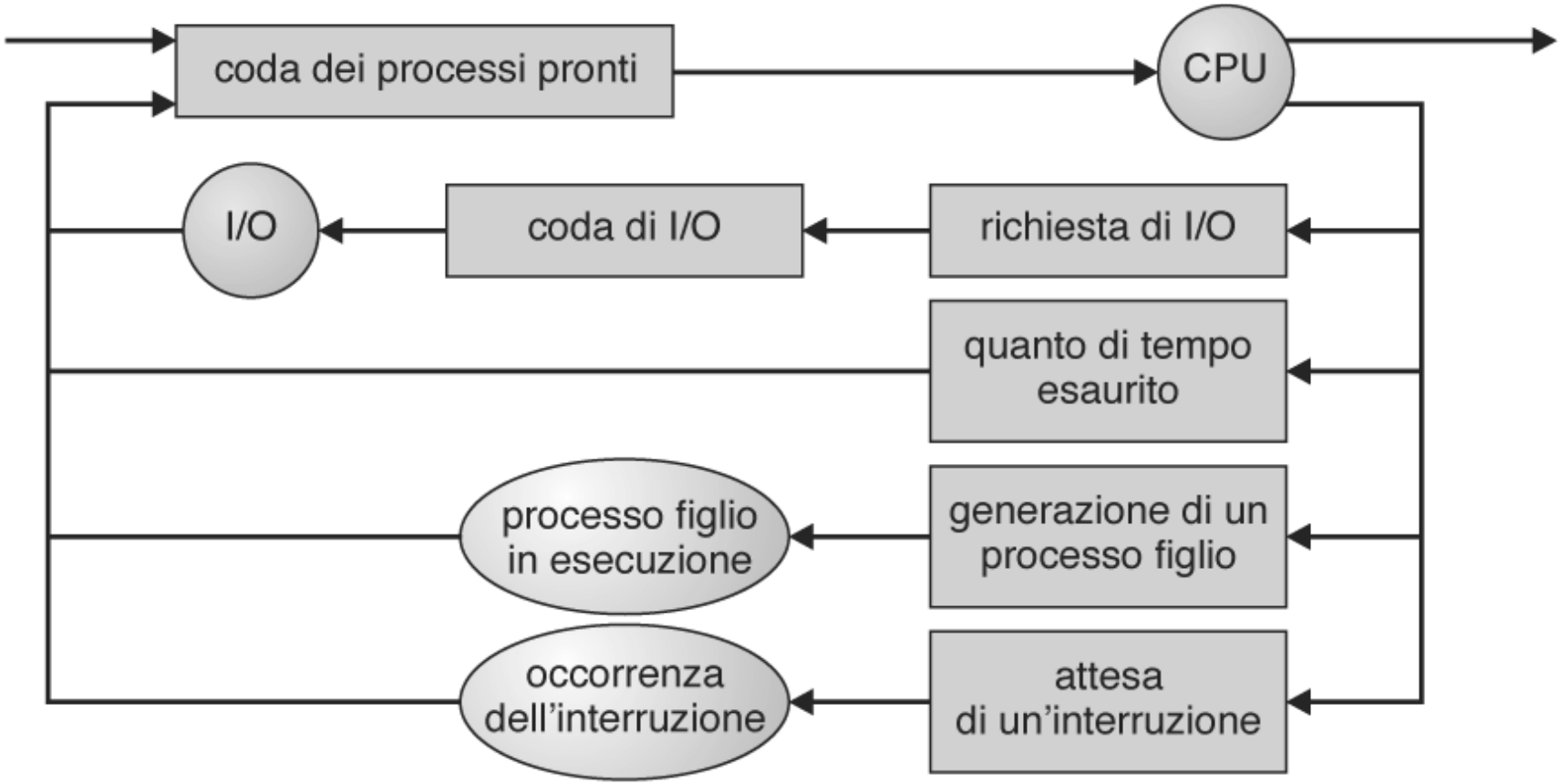
Supporti per la gestione dei processi (2)

- Code dei processi nei vari stati
- Ogni coda a due puntatori
  - Primo PCB
  - Ultimo PCB
- Ogni PCB ha un puntatore al successivo



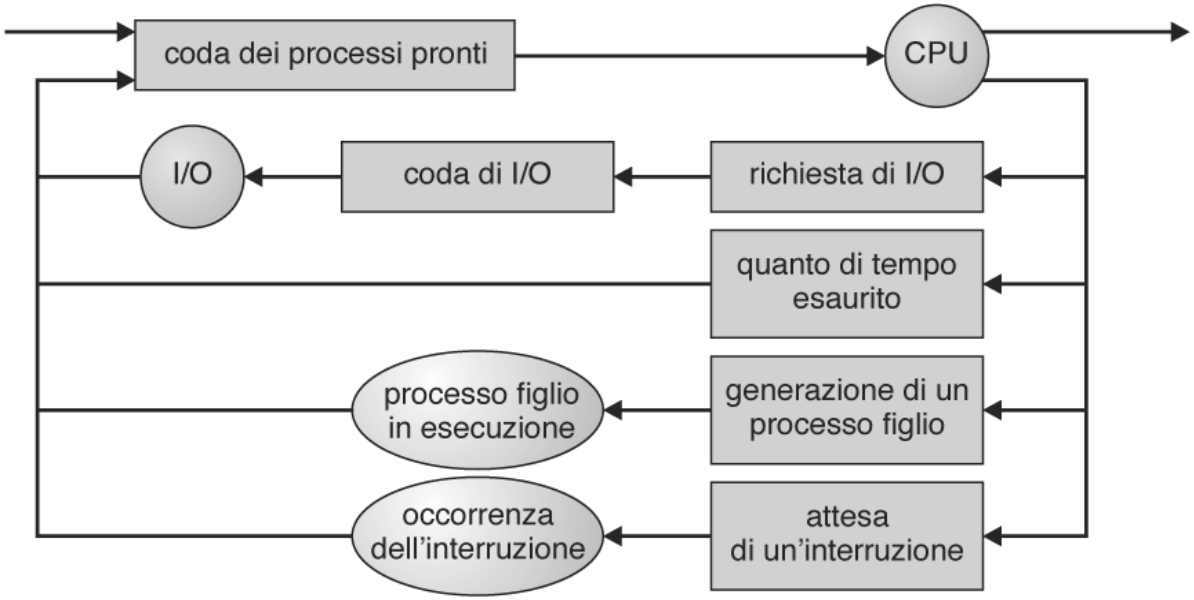
Supporti per la gestione dei processi (3)

- Diagramma di accodamento: descrive lo scheduling



Supporti per la gestione dei processi (4)

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

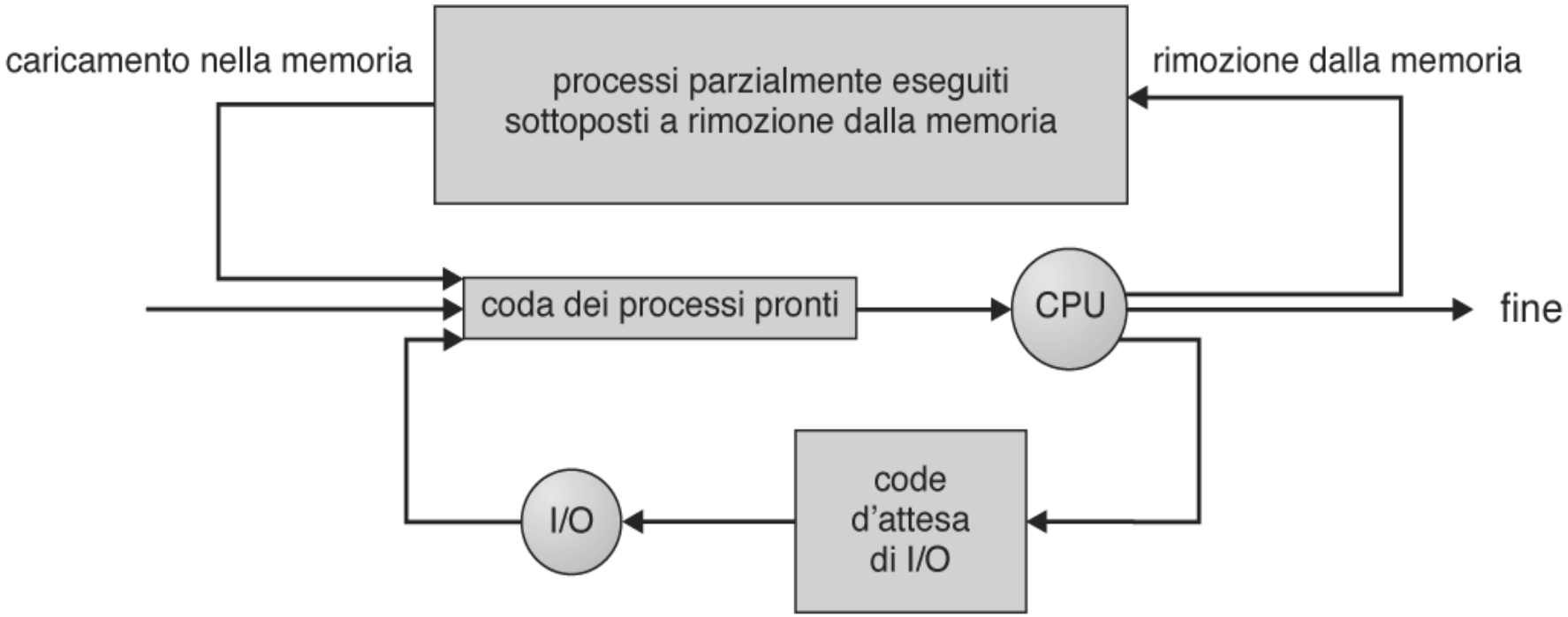


Gli schedulatori (1)

- **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
  - Stabile se velocità di creazione processi = velocita terminazione processi
  - 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): 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

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**
  - La stabilità dipende da limiti fisici e dalla disciplina dell'utente

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
  - Se sbilanciato coda pronti sempre vuota (prevalenza I/O bound) o coda I/O sempre vuota e quindi dispositivi inutilizzati (prevalenza CPU 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 necessità più memoria di quella disponibile

Il cambio di contesto (1)

context switch

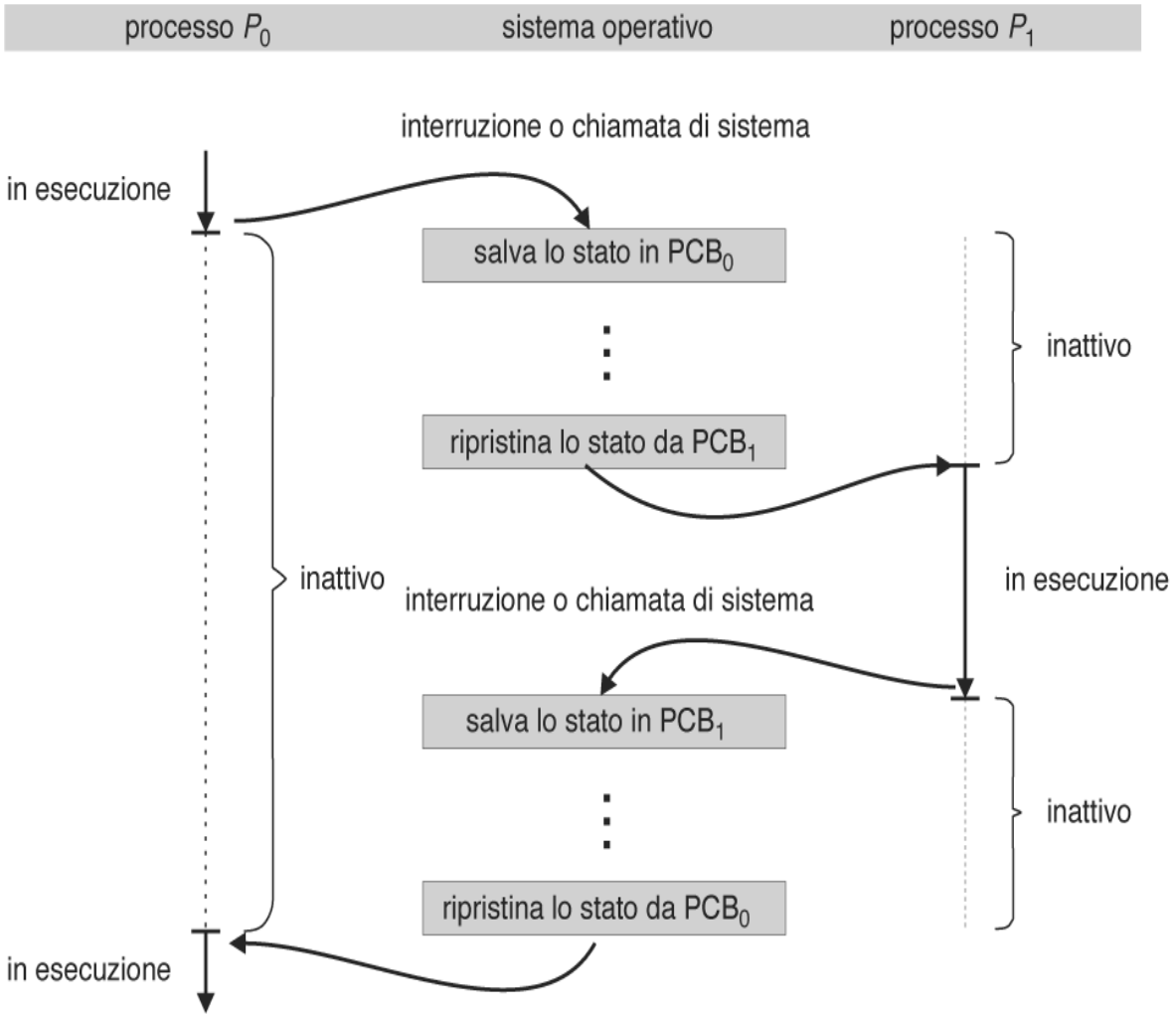
- E' il **passaggio della CPU da un processo ad un altro**

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

- Il tempo per il cambio di contesto è puro **tempo di gestione del sistema**
  - Poiché durante il cambio non vengono compiute operazioni utili per la computazione dei processi (calo delle prestazioni)
- I tempi per i cambi di contesto dipendono sensibilmente dal supporto hardware (e.g. registri disponibili e/o registri dedicati)
  - tipicamente inferiore ai 10 millisecondi

Il cambio di contesto (2)

In cosa consiste il context switch?



Processi come flussi di operazioni

- Processo = flusso di esecuzione di computazione

- Flussi separati = processi separati
- I prossimi argomenti riguarderanno
  - Come si generano i processi
  - 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)

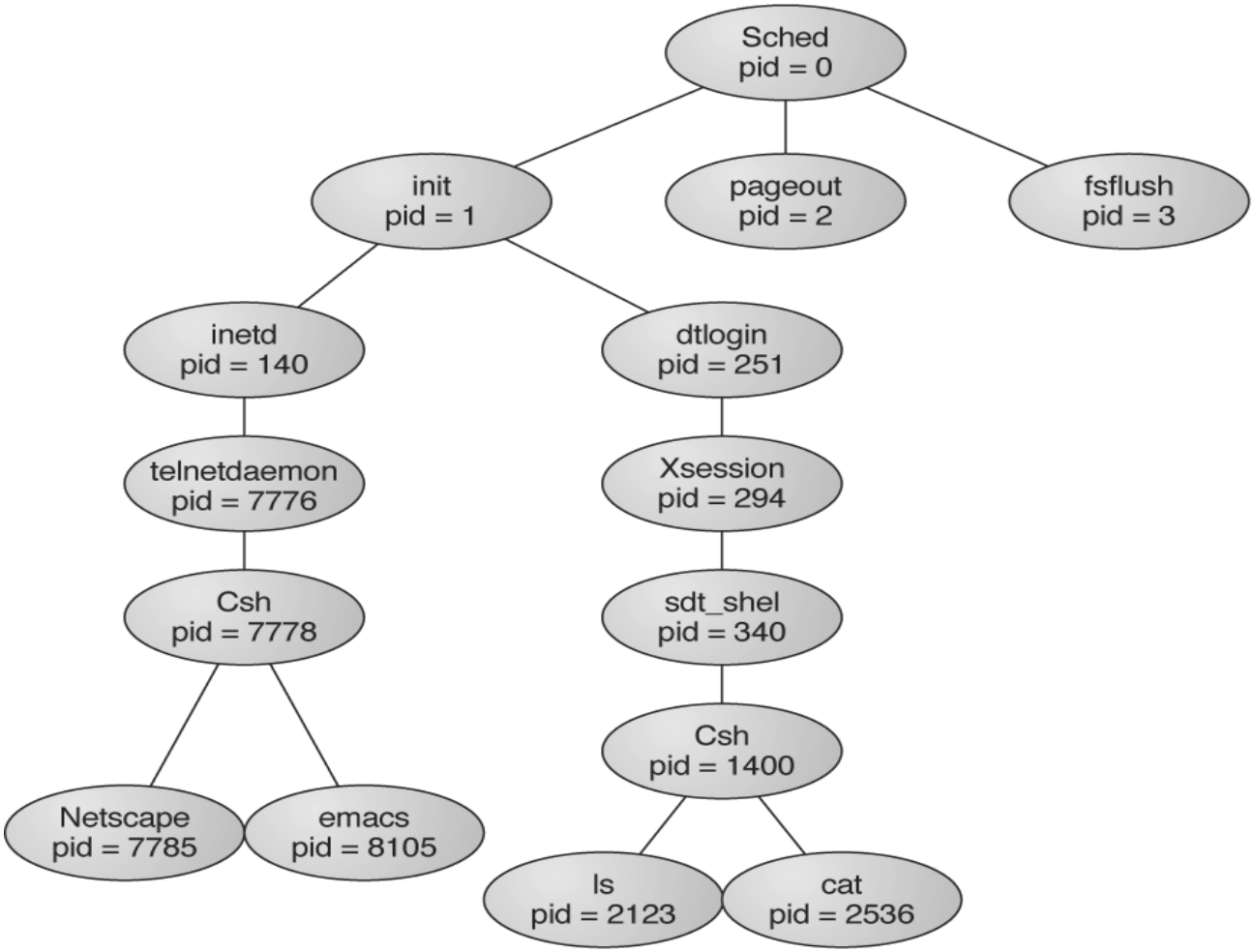
Creazione (o generazione) di un processo (1)

- Un processo in esecuzione può creare numerosi sottoprocessi usando un'apposita chiamata di sistema (create_process)
  - Processo generante → processo padre
  - Processo generato → processo figlio
- In unix si usa la chiamata fork()
- Si crea un albero di processi (visibile ad es. tramite ps -el)
- Ogni processo ha un identificatore univoco PID (intero)
- Esecuzione del processo padre dopo la creazione del figlio:
  - Continua in modo concorrente, oppure
  - Attende la terminazione del figlio

Albero dei processi

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

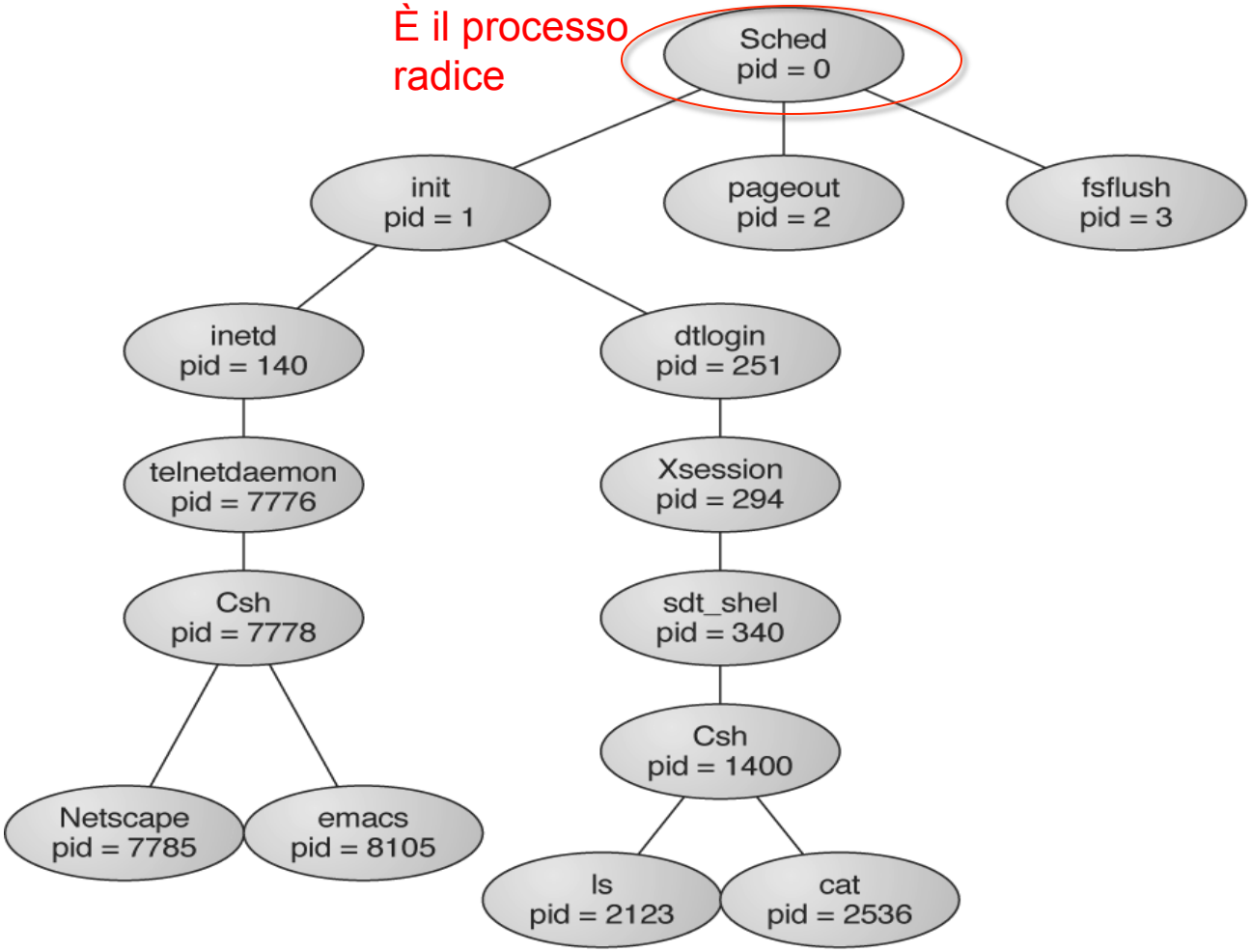
Es. albero dei processi in un sistema Solaris



Albero dei processi

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

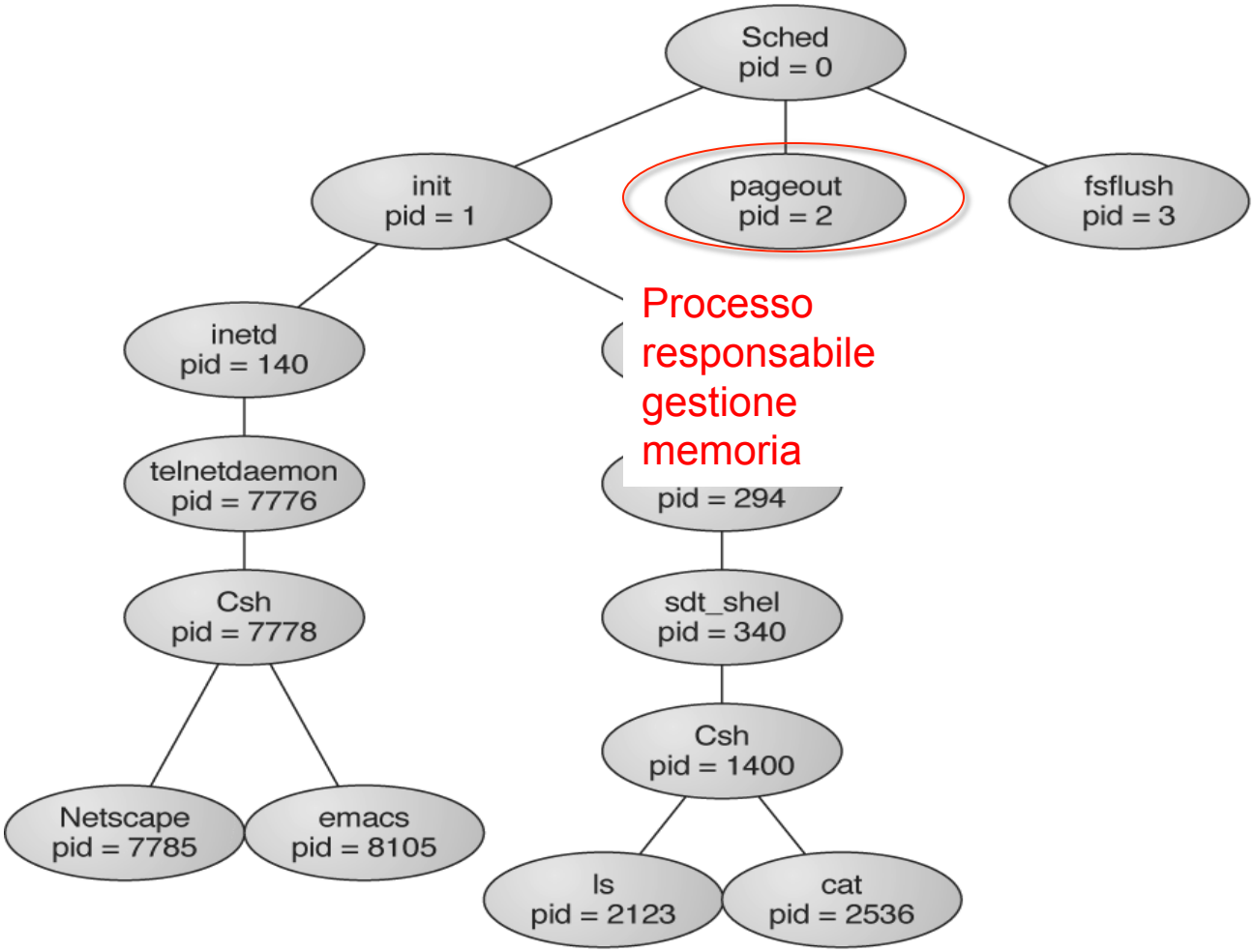
Es. albero dei processi in un sistema Solaris



Albero dei processi

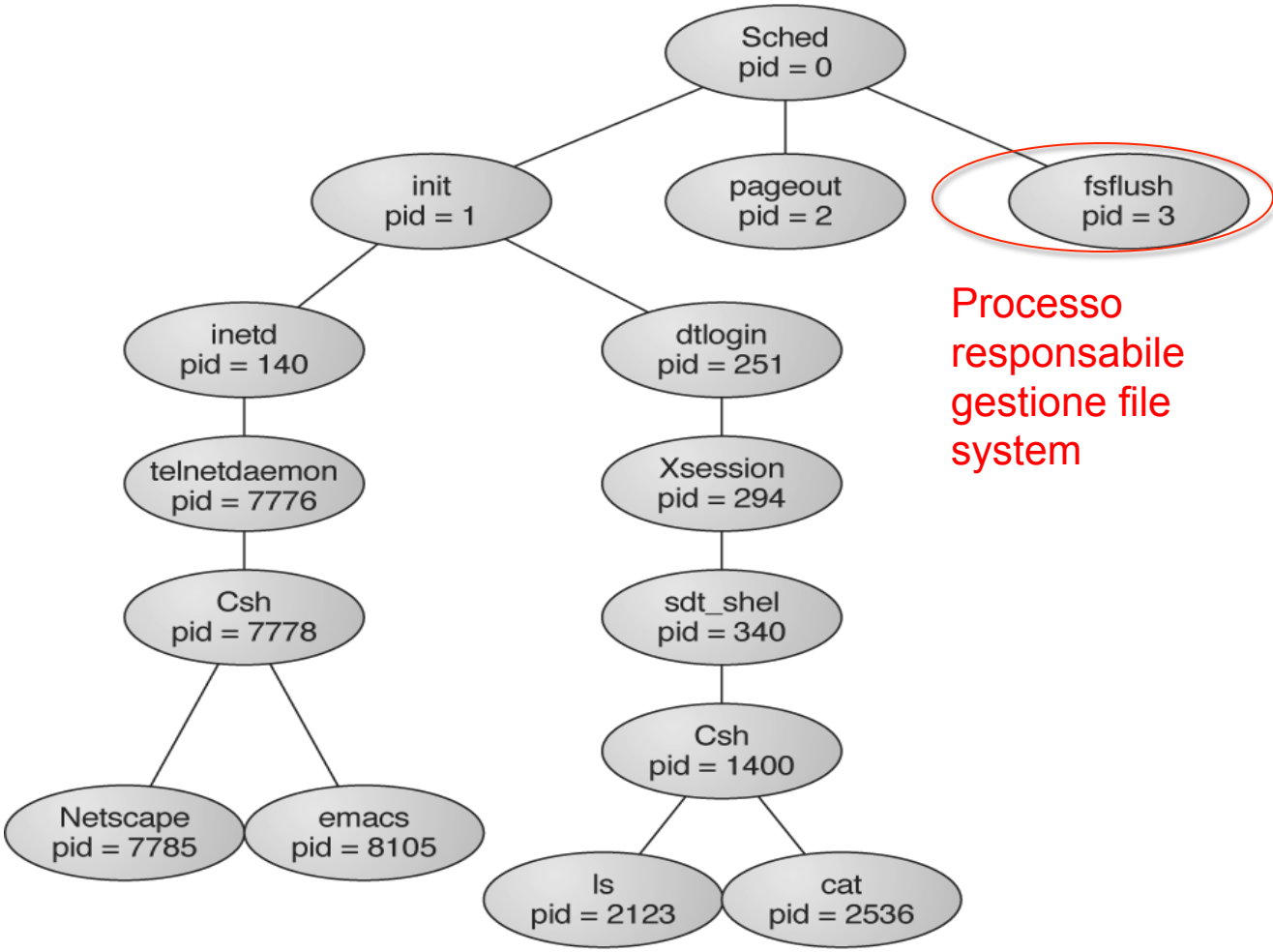
Il processo padre crea processi figli, i quali a loro volta creano altri processi formando un albero di processi

Es. albero dei processi in un sistema Solaris



Albero dei processi

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



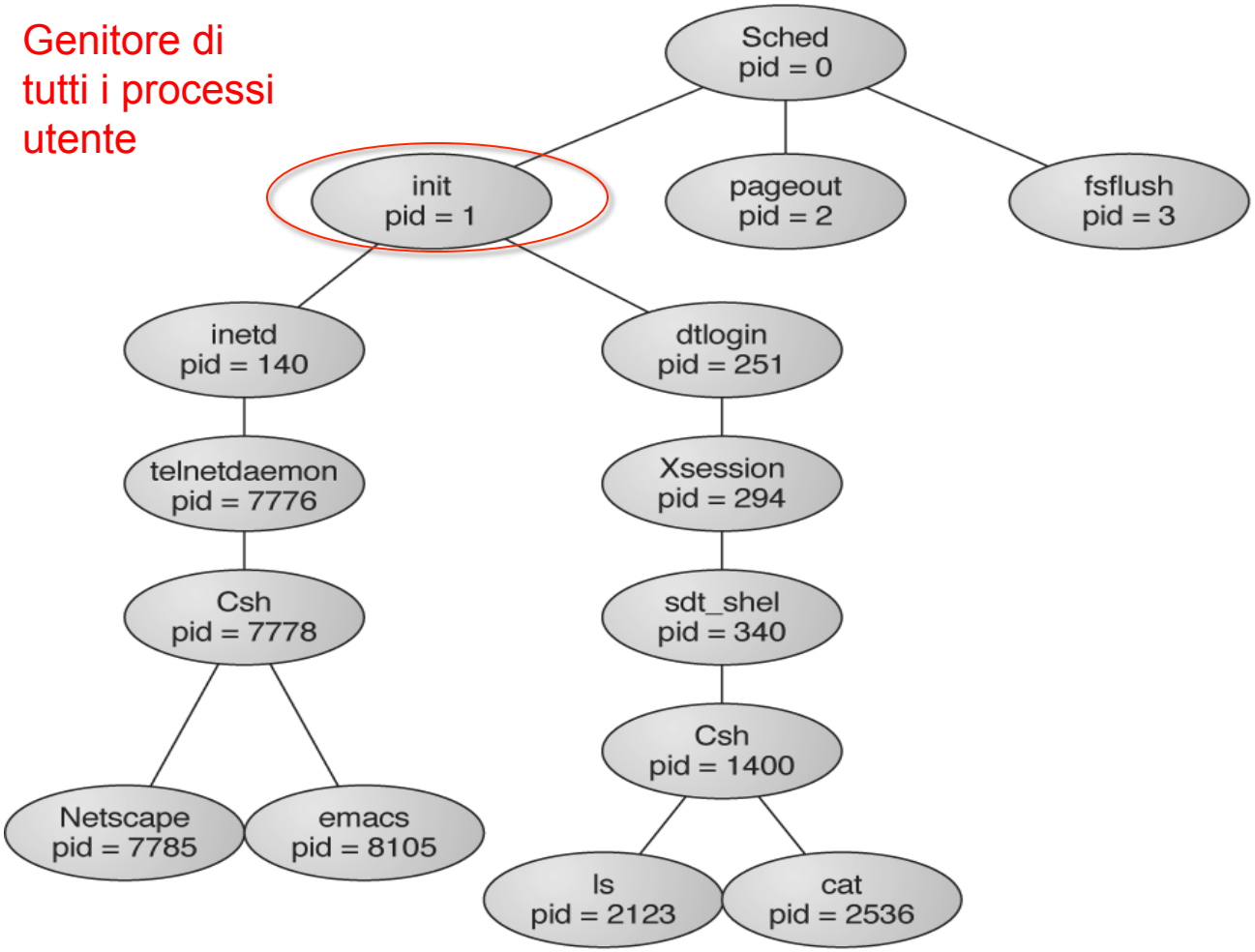
Processo responsabile gestione file system

Es. albero dei processi in un sistema Solaris

Albero dei processi

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

Genitore di tutti i processi utente

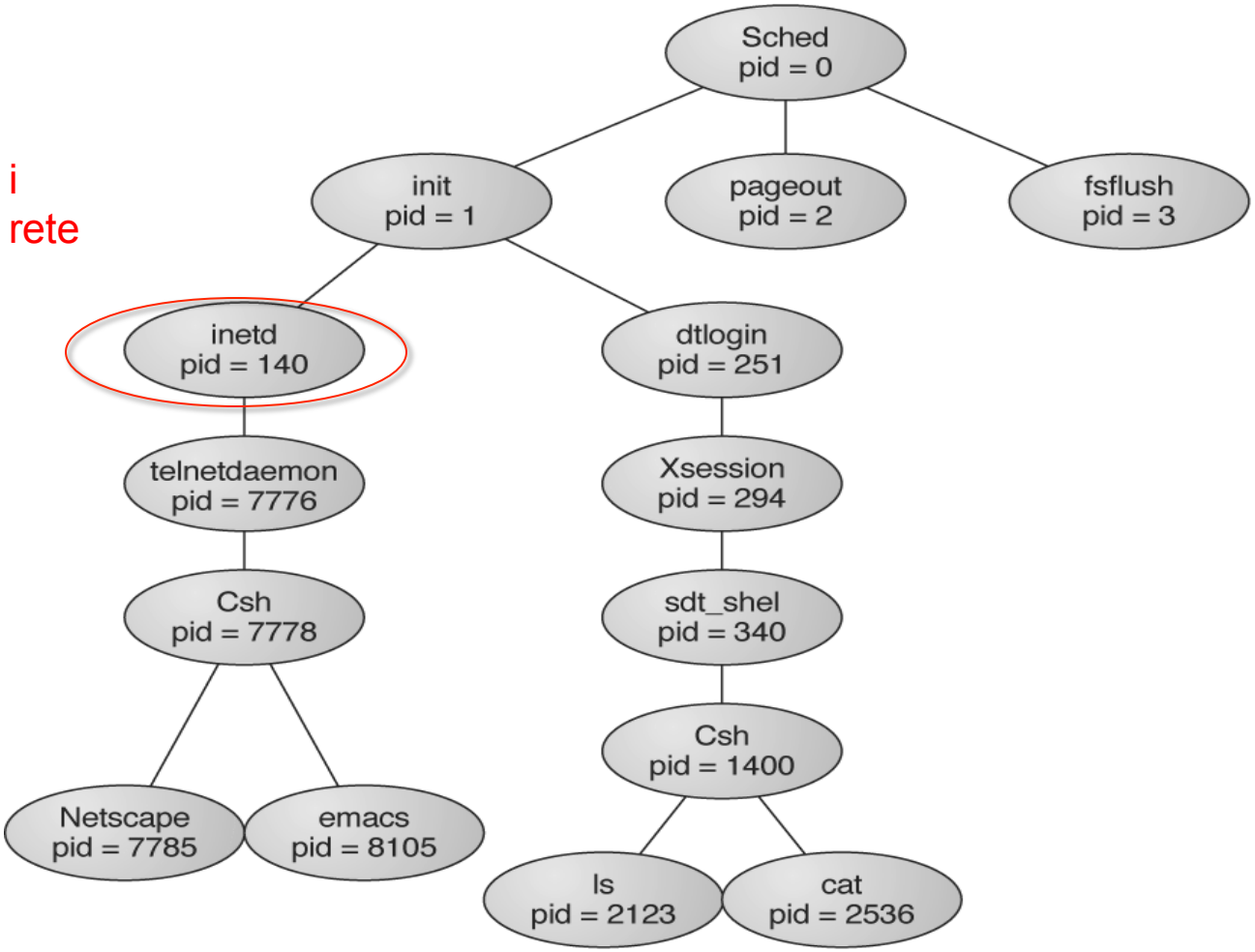


Es. albero dei processi in un sistema Solaris

Albero dei processi

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

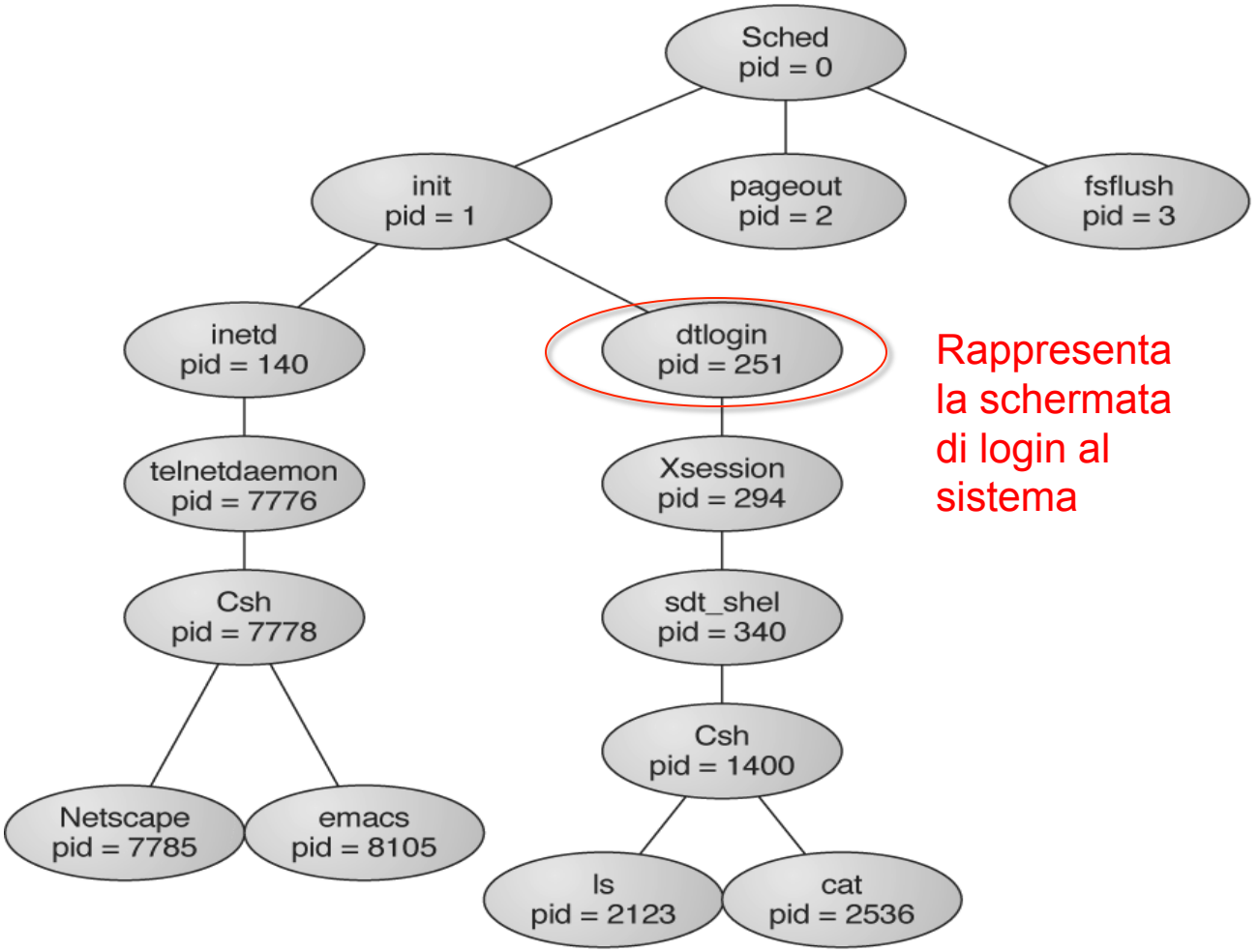
Gestisce i servizi di rete



Es. albero dei processi in un sistema Solaris

Albero dei processi

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



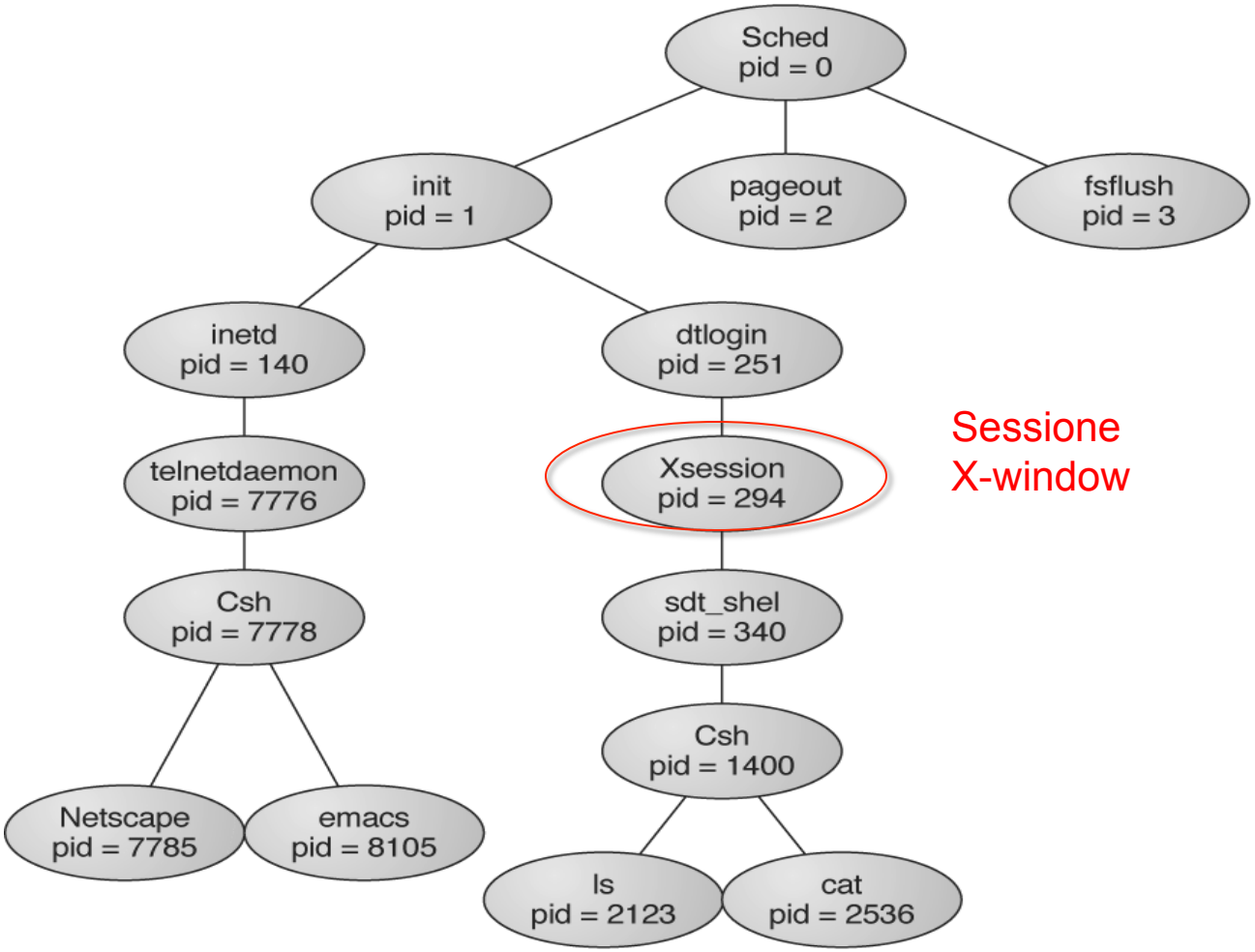
Rappresenta la schermata di login al sistema

Es. albero dei processi in un sistema Solaris

Albero dei processi

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

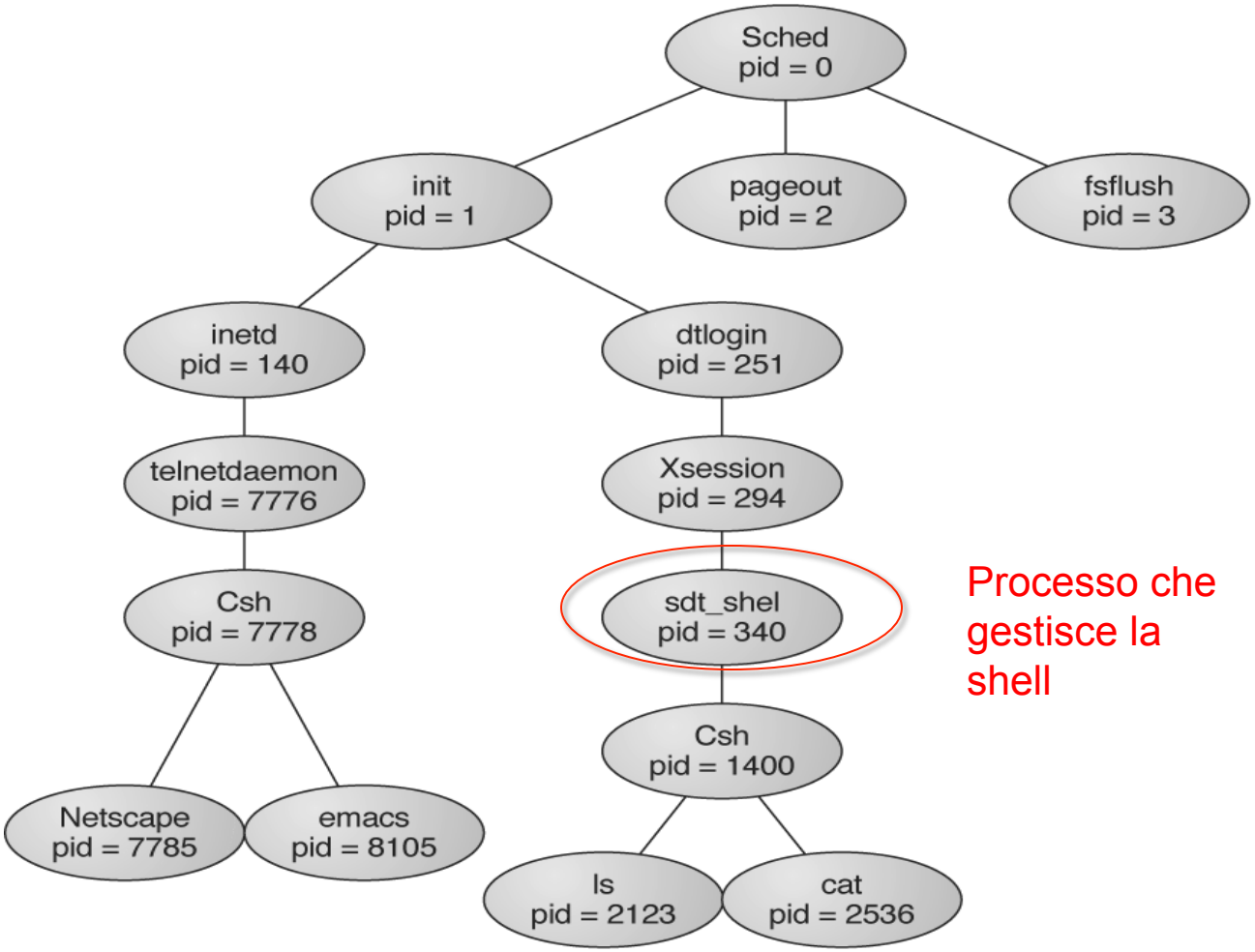
Es. albero dei processi in un sistema Solaris



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 Solaris

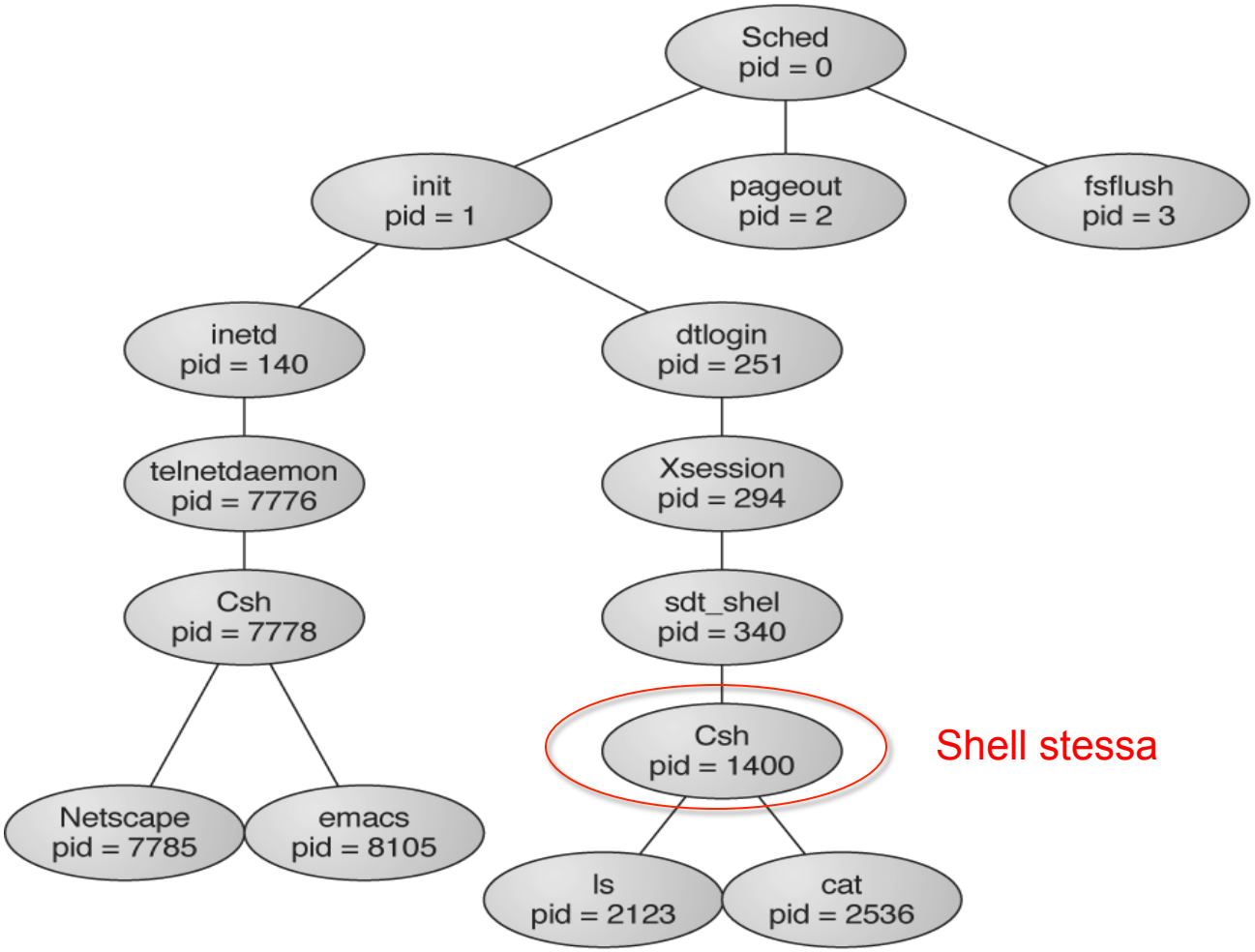


Processo che gestisce la shell

Albero dei processi

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

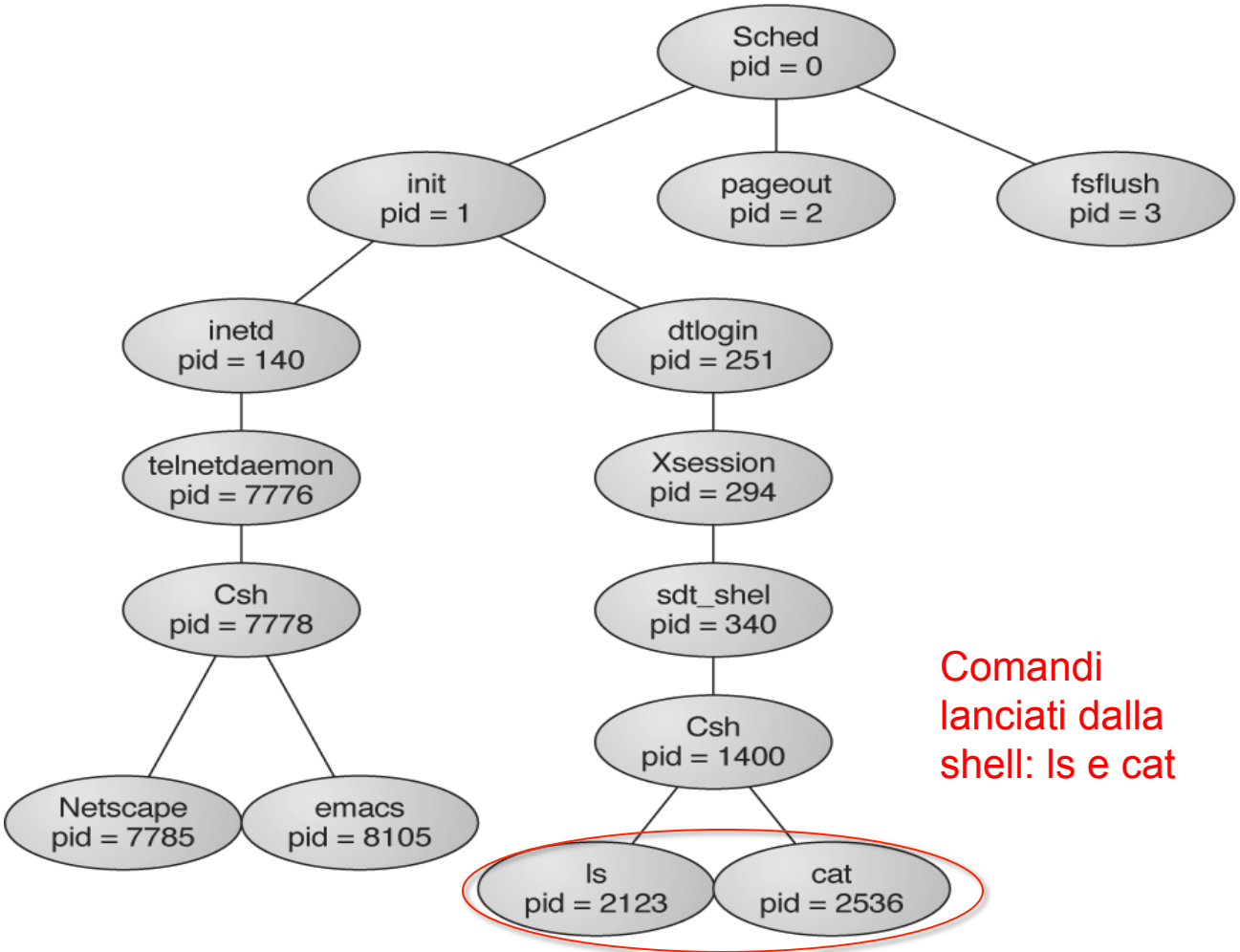
Es. albero dei processi in un sistema Solaris



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 Solaris

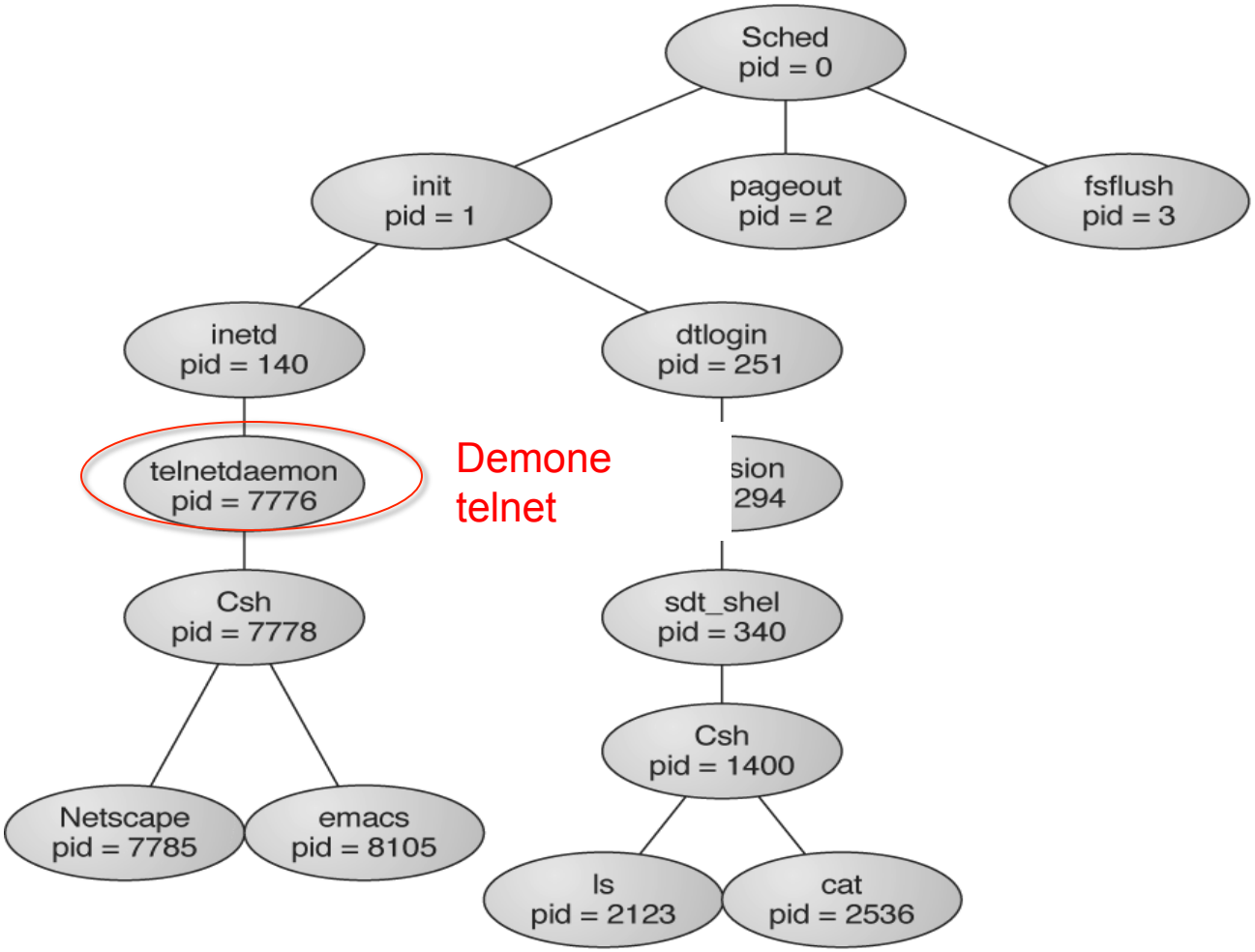


Comandi lanciati dalla shell: ls e cat

Albero dei processi

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

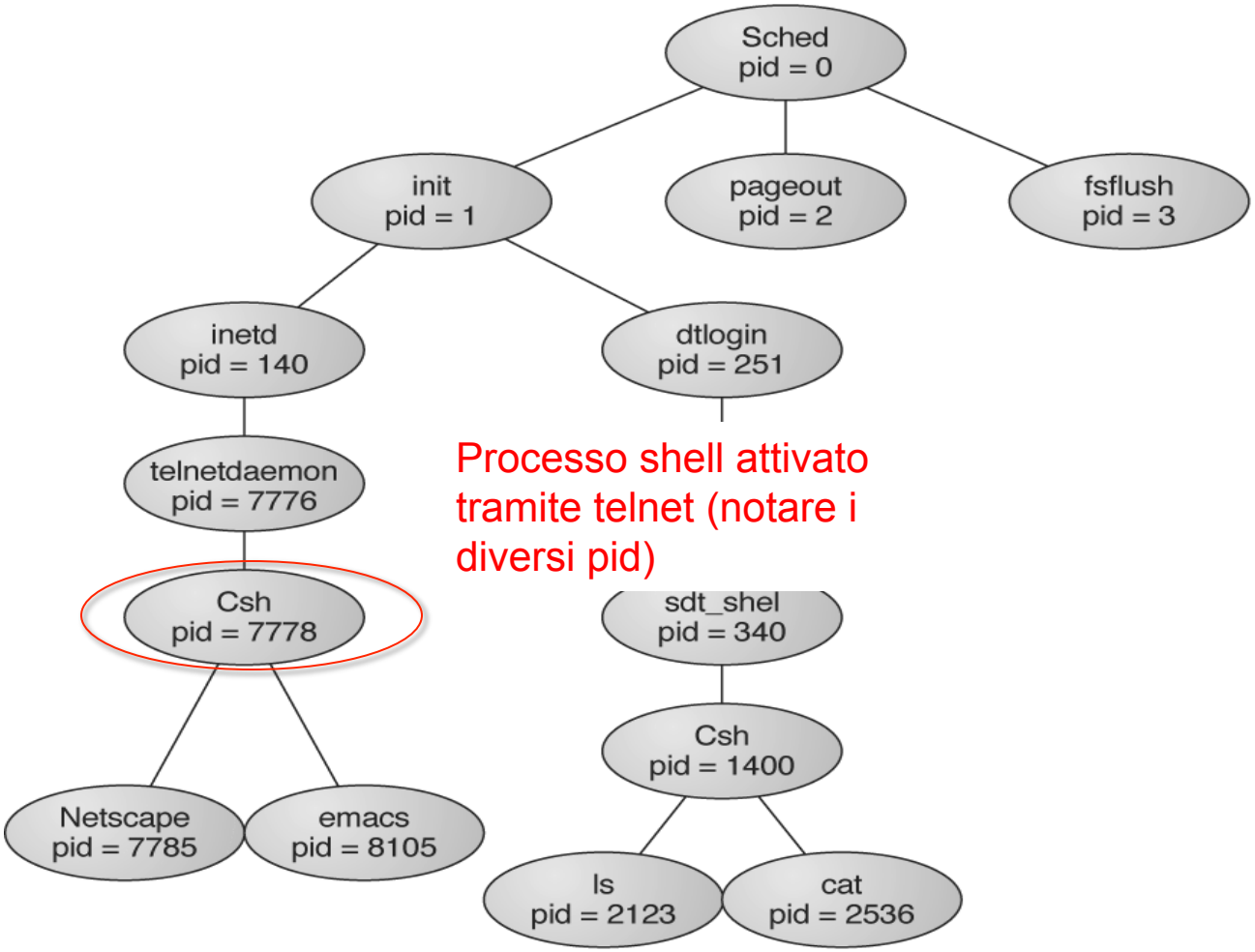
Es. albero dei processi in un sistema Solaris



Albero dei processi

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

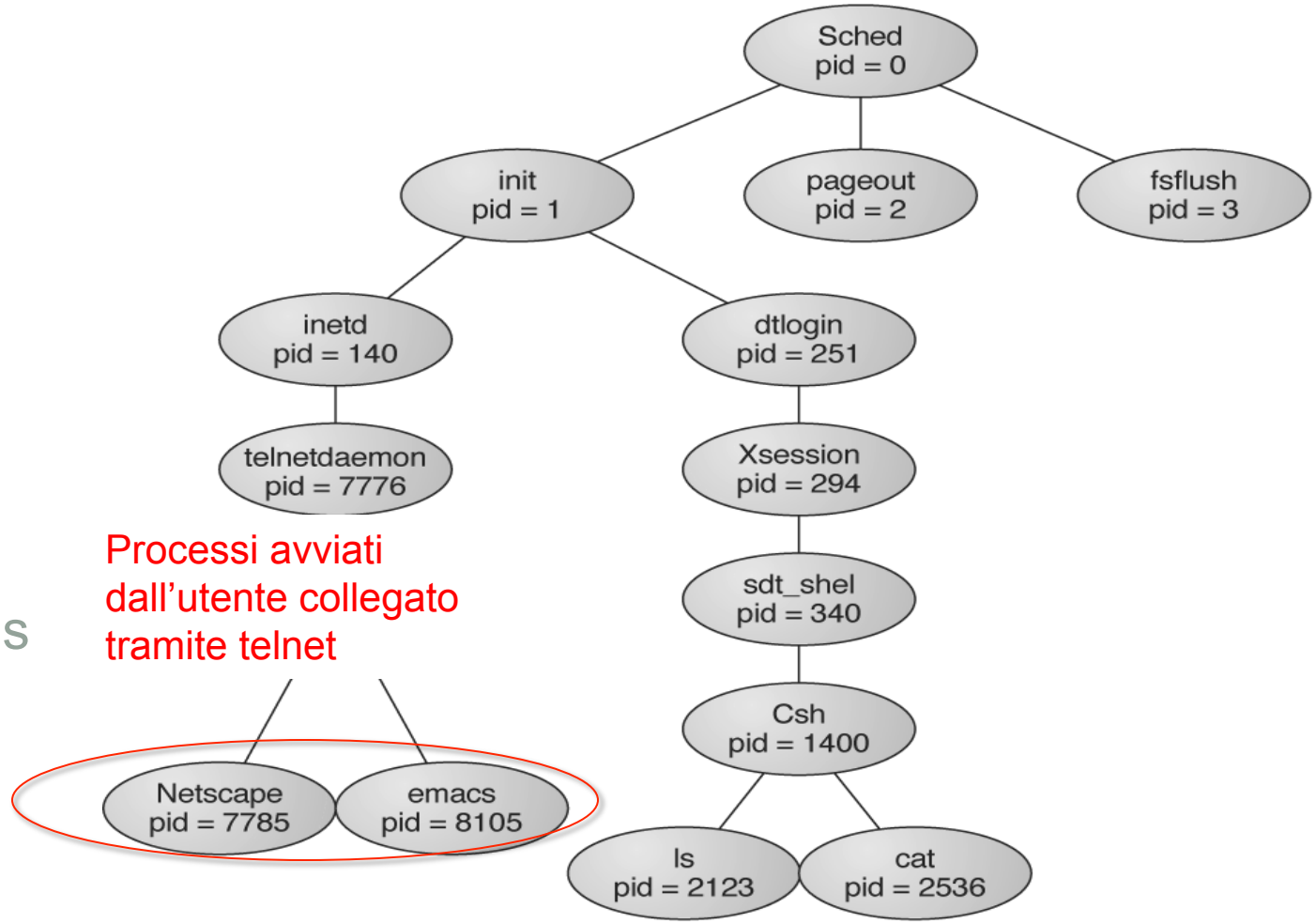
Es. albero dei processi in un sistema Solaris



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 Solaris



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

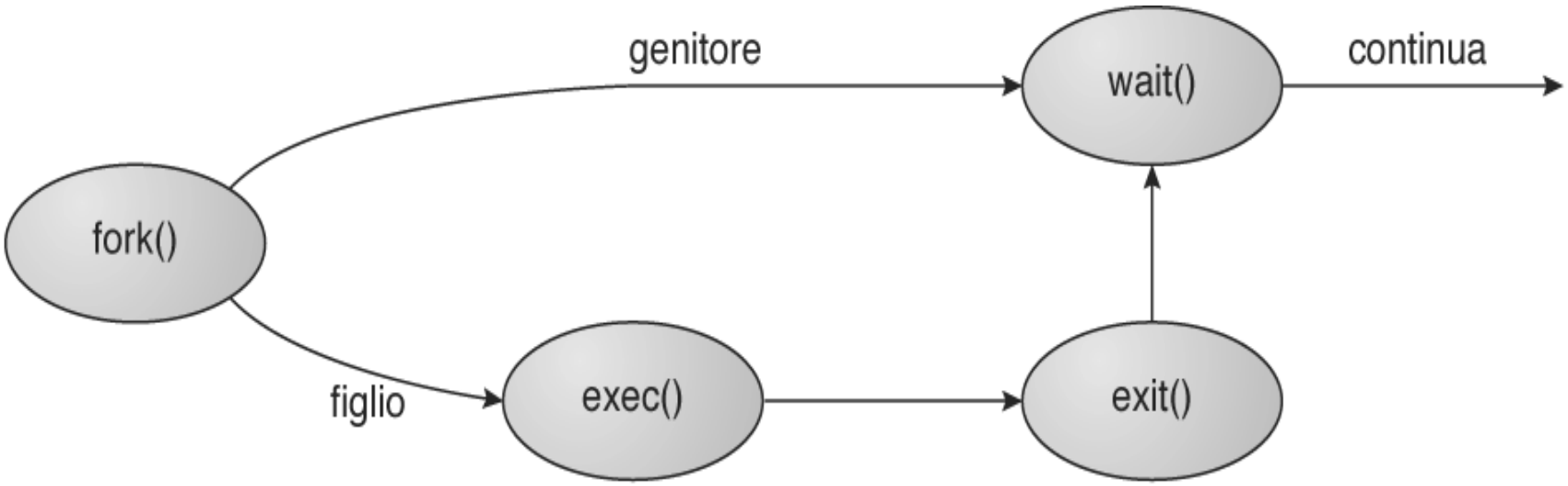
- Inoltre alla creazione un processo figlio riceve dal padre i dati di inizializzazione
  - Parametri utili per l'esecuzione del processo
  - E.g. un processo che permette di visualizzare un immagine riceverà dal padre il percorso del file

Spazio di indirizzamento (1)

- Lo spazio di indirizzamento (memoria dedicata al processo) 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 della chiamata fork() e wait() (1)

```
1. #include <pthread.h>
2. #include <stdio.h>
3. #include <unistd.h>
4. #include <sys/types.h>

5. int main()
6. {
7.     pid_t pid;

8.     // fork a child process
9.     pid = fork();

10.    if (pid < 0) { // error occurred
11.        fprintf(stderr, "Fork Failed\n");
12.        exit(-1);
13.    }
14.    else if (pid == 0) { // child process
15.        printf("I am the child %d\n",pid);
16.        execlp("/bin/ls","ls",NULL);
17.    }
18.    else { // parent process
19.    // parent will wait for the child to complete
20.        printf("I am the parent %d\n",pid);
21.        wait(NULL);
22.
23.        printf("Child Complete\n");
24.        exit(0);
25.    }
26. }
```

Esempio in Unix della chiamata fork() e wait() (2)

```
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);
  }
}
```

- La chiamata `fork()` crea un nuovo processo
- Il valore di ritorno vale
  - 0 per il figlio
  - Diverso da 0 per il padre
- La fork crea una copia dello spazio degli indirizzi del genitore
- Entrambi, padre e figlio, continuano l'esecuzione dalla chiamata successiva alla fork
- Il padre entrerà nel secondo else mentre il figlio nel primo

Esempio in Unix della chiamata fork() e wait() (3)

```
int main()
{
  …
  …

  else if (pid == 0) { // child process
      printf("I am the child %d
         \n",pid);
      execlp("/bin/ls","ls",NULL);
  }

  …
  …
}
```

- La chiamata di sistema **`exec()`** viene tipicamente eseguita dopo la fork
- Essa sovrascrive lo spazio degli indirizzi del chiamante caricando un nuovo programma
- L'esecuzione della chiamata **`execlp()`** (specializzazione di **`exec`**) carica in memoria il programma **`"/bin/ls"`**
- Cosa farà il processo figlio?

Esempio in Unix della chiamata fork() e wait() (4)

```
int main()
{
  …
  …

  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);
  }
}
```

- 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 specifico figlio
- Cosa fa il processo padre?

- Vediamo l'output dell'esecuzione
- Il codice può essere compilato da linea di comando usando **`gcc`**

Esempio 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 contenenti info sui processi
- ZeroMemory alloca la memoria per il nuovo processo
- CreateProcess() è il corrispettivo di fork()
  - Riceve molti parametri
- WaitForSingleObject() è il corrispettivo di wait()
  - Riceve come parametro il processo da attendere

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

```
    }
    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);
    }
}
```

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

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

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 () { }
```