Introduzione ai Processi

Sommario:

- Definizioni: Programma, Processo, File Binario Eseguibile, Applicazione.
- Formato dei file binari eseguibili
- Process Control Block
- Creazione di Processo
- Processo Padre e Processi Figli
- Attesa Terminazione Processo Figlio
- Processi Zombie
- Gestione asincrona della Terminazione di Processo Figlio

NOZIONI DI PROGRAMMA e PROCESSO

Il termine **PROGRAMMA** è ambiguo poiché può indicare:

1. Descrizione di un algoritmo mediante una **sequenza di istruzioni** espresse in un qualche linguaggio.

2. Un **FILE BINARIO ESEGUIBILE**, collocato su disco, che contiene
   - una sequenza di istruzioni macchina da far eseguire
   - e **tutte le informazioni necessarie** affinché il sistema operativo possa predisporre il contesto INIZIALE di esecuzione in memoria di quelle istruzioni.

   L'eseguibile ha perciò un formato noto al sistema operativo.
   Si specifica "contesto INIZIALE" poiché il contesto varia durante l'esecuzione, in quanto le istruzioni eseguite modificano il contesto stesso.

3. Un **PROCESSO**, ovvero l'istanziazione in memoria (immagine in memoria) di un file binario eseguibile dopo che questo è stato posto in esecuzione dal loader del sistema operativo.
   - Il processo è una entità dinamica, che può modificare se stesso nel tempo, addirittura può duplicarsi.

4. Una **APPLICAZIONE**, cioè un insieme di processi, generalmente creati da un unico eseguibile iniziale, che eseguono per raggiungere un obiettivo comune.

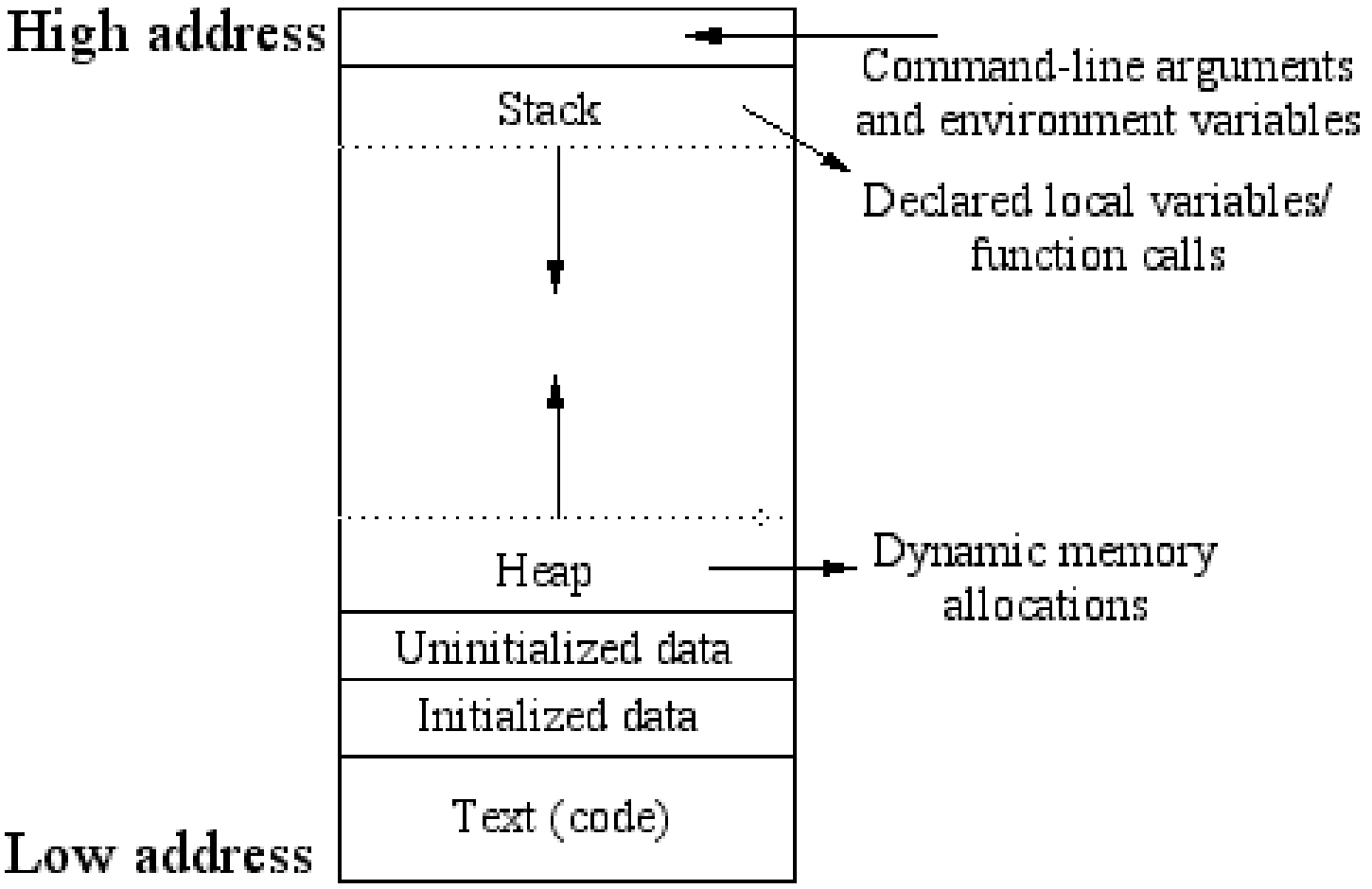
PROGRAMMA vs. PROCESSO

- Un programma in esecuzione prende il nome di **processo**
  - entità ‘attiva’, la cui esecuzione è ad opera di un processore (la CPU) e la cui struttura e attività sono descritte da un certo programma o porzione di programma
  - a differenza del processo, il relativo programma è un'entità passiva, come rappresentazione (o insieme di istruzioni) interpretabili dall'esecutore del processo stesso
  - quasi sinonimi ma non troppo: *task*, *job*

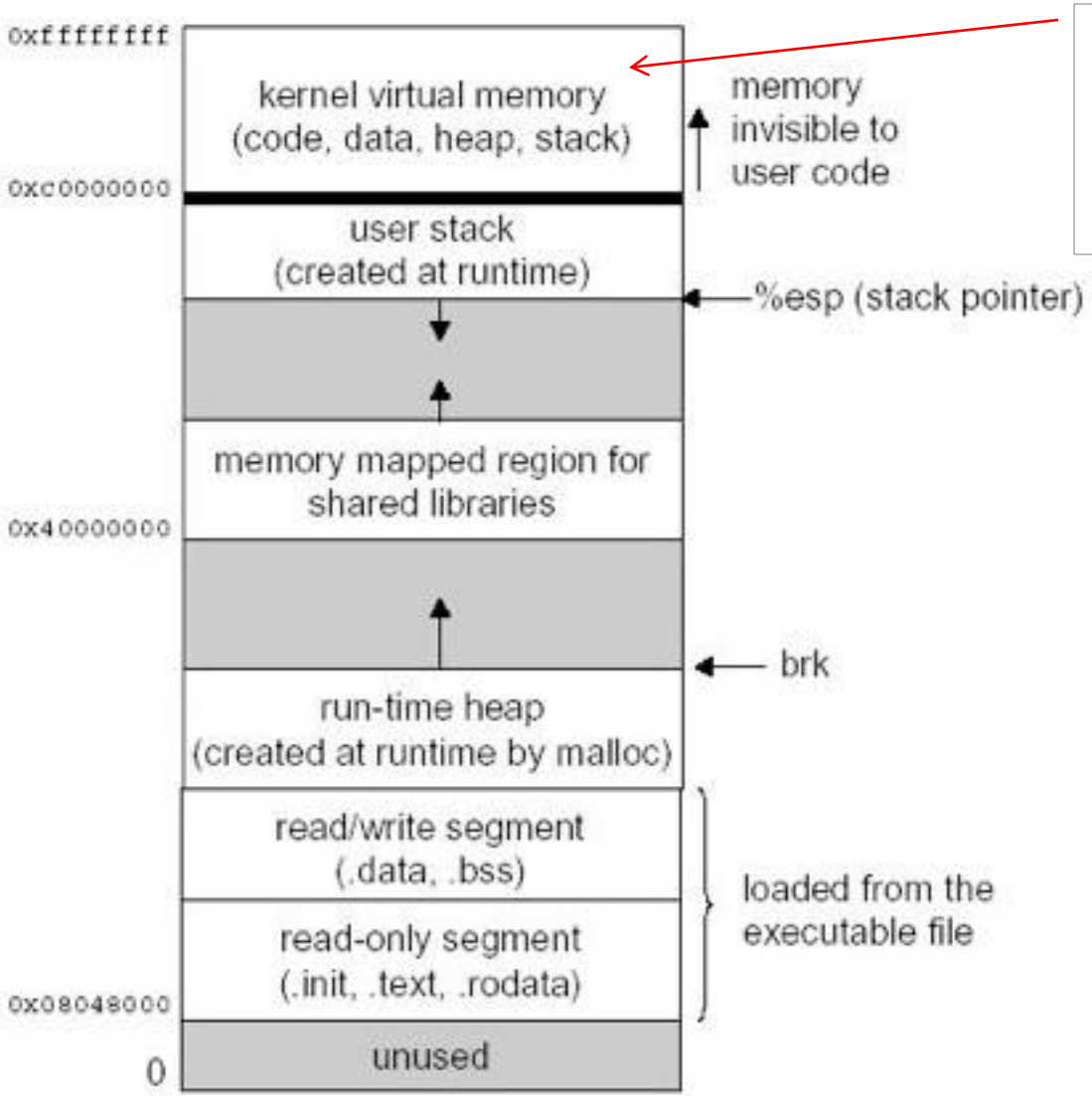
- Generalmente c’è una corrispondenza 1:1 fra *applicazioni* e processi
  - ogni applicazione eseguita su un computer è un processo
  - in alcuni casi tuttavia una medesima applicazione può lanciare più processi

- Ogni processo è identificato a tempo di esecuzione da un **PID** (process identifier) assegnatogli dal sistema operativo
  - numero intero monotono crescente

MEMORY LAYOUT DI UN PROCESSO IN UNIX



LAYOUT PROCESSO IN LINUX



user area
appartiene al processo, ma è utilizzabile solo con la CPU in kernel mode

PROCESSI E FORMATI DEI FILE ESEGUIBILI

- Un file binario *eseguibile* contiene un programma che può essere caricato in memoria e mandato in esecuzione come processo

- Esistono **formati** diversi per i file binari eseguibili, proposti dalle diverse famiglie dei sistemi operativi

  – su sistemi UNIX/Linux esempi sono **ELF** (recente), **a.out** (storico)
    - ELF è il default su sistemi Linux

  – su sistemi Windows **PE** (Portable Executable)

- In generale un formato definisce un layout con cui sono organizzate le informazioni all'interno del file eseguibile
  – dati, codice e meta-dati

FORMATO a.out

- **a.out** is a file format used in **older versions** of Unix-like computer operating systems for executables, object code, and, in later systems, shared libraries.
  - the name stands for *assembler output.*
  - a.out remains the default output file name for executables created by certain compilers/linkers when no output name is specified
    - even though these executables are no longer in the a.out format
- Several variants
  - OMAGIC
    - had contiguous segments after the header, with no separation of text and data. This format was also used as object file format.
  - NMAGIC
    - similar to OMAGIC, however the data segment is loaded on the immediate next page after the end of the text segment, and the text segment was marked read-only.
  - ZMAGIC
    - adds support for demand paging. The length of the code and data segments in the file had to be multiples of the page size.
  - QMAGIC
    - binaries loaded one page above the bottom of the virtual address space, in order to permit trapping of null pointer dereferences via a segmentation fault

a.out LAYOUT

- An a.out file consists of up to seven sections, in the following order:
  - **exec header**
    - contains parameters used by the kernel to load a binary file into memory and execute it, and by the link editor ld to combine a binary file with other binary files. This section is the only mandatory one.
  - **text segment**
    - contains machine code and related data that are loaded into memory when a program executes. May be loaded read-only.
  - **data segment**
    - contains initialized data; always loaded into writable memory.
  - **text relocations**
    - contains records used by the link editor to update pointers in the text segment when combining binary files.
  - **data relocations**
    - like the text relocation section, but for data segment pointers.
  - **symbol table**
    - contains records used by the link editor to cross-reference the addresses of named variables and functions (symbols) between binary files.
  - **string table**
    - contains the character strings corresponding to the symbol names.

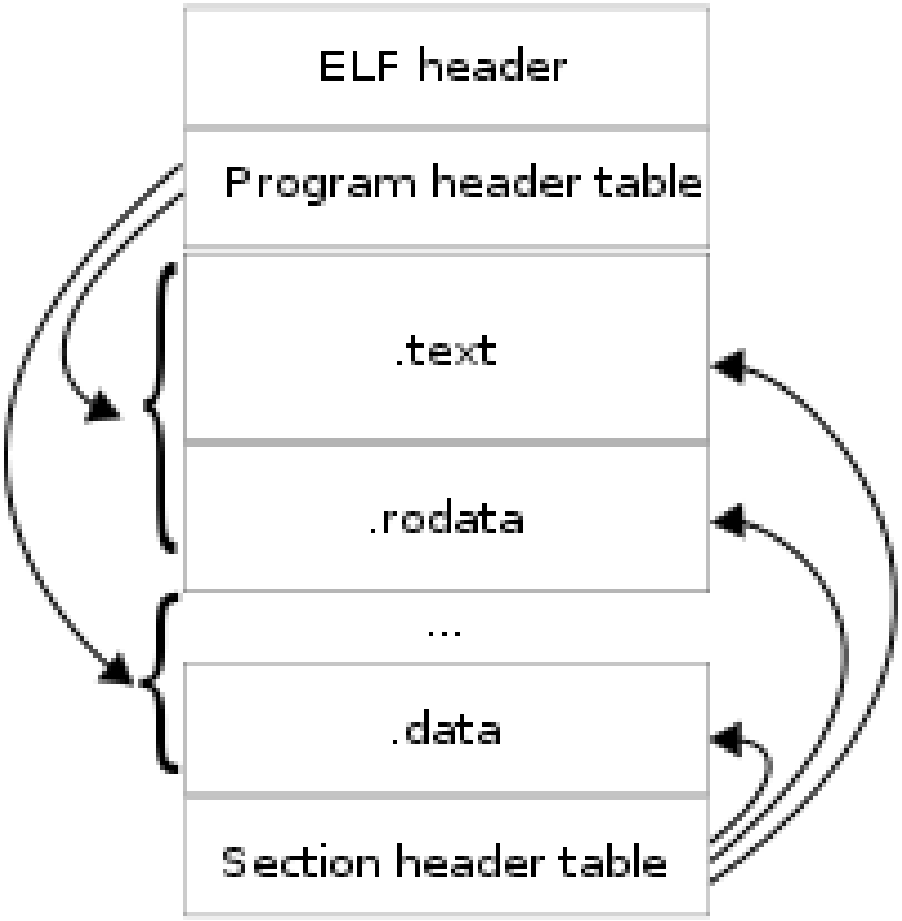
EXECUTABLE AND LINKING FORMAT (**ELF**)

- Common standard file format for executables, object code, shared libraries, and core dumps.
  - not bound to any particular processor or architecture.
  - adopted by many different operating systems on many different platforms
    - has replaced older executable formats such as a.out and COFF in many Unix-like operating systems such as Linux, Solaris,FreeBSD
    - some adoption in non-Unix operating systems
      - Itanium version of OpenVMS, BeOS, PlayStation
  - extensible

- Utility to analyse an executable file under UNIX env
  - readelf, objdump, file , ldd

ELF LAYOUT

- ELF header
  - describes where the various parts are located in the file
- File data
  - **Program** header table, describing zero or more *segments*
    - contain information that is necessary for runtime execution of the file
  - **Section** header table, describing zero or more *sections*
    - contain important data for linking and relocation
  - Data referred to by entries in the program header table, or the section header table



Interprete per il caricamento degli eseguibili ELF

– La **Program** header table, contiene informazioni necessarie al momento dell'esecuzione del file

– In particolare, nella sezione INTERP, della Program header table di un file eseguibile, troviamo l'informazione che dice al sistema operativo quale è l'interprete che dobbiamo utilizzare per comprendere il contenuto del file stesso e per effettuare il caricamento in memoria dell'eseguibile per poi eseguirlo.

L'interprete si occuperà di:

–creare l'immagine del processo in memoria,
–caricare le librerie condivise necessarie all'eseguibile e elencate nell'eseguibile stesso,
–passare il controllo all'eseguibile stesso saltando all'indirizzo della istruzione di partenza dell'eseguibile stesso.

Per eseguibili a 64 bit, per l'architetture x86 a 64bit,solitamente l'interprete è chiamato ld-linux-x86_64.so.2

Verificare l'interprete per un eseguibile eseguendo: **readelf -l** nomefile
e leggendo nell'output il contenuto della sezione INTERP:

```
INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
               0x000000000000001c 0x000000000000001c  R      1
               [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

Ad esempio, provare ad eseguire readelf -l /bin/grep

PORTABLE EXECUTABLE (PE)

- File format for executables, object code, and DLLs, used in 32-bit and 64-bit versions of Windows operating systems
  - the term "portable" refers to the format's versatility in numerous environments of operating system software architecture.
- Encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code.
  - including dynamic library references for linking, API export and import tables, resource management data and thread-local storage (TLS) data.
  - on NT operating systems, the PE format is used for EXE, DLL, OBJ, SYS (device driver), and other file types.
- PE is a modified version of the Unix COFF file format.
  - PE/COFF is an alternative term in Windows development.

- **Layout**
  - consists of a number of headers and sections that tell the dynamic linker how to map the file into memory
- Recently extended by Microsoft's .NET Framework with features to support the Common Language Runtime.

COMPONENTI DI UN PROCESSO

- Le componenti principali di un processo in un S.O. moderno sono:
  - **codice** del programma
  - **contesto di esecuzione**
    - immagine dei registri del processore
    - program counter (PC)
  - **stack**
  - area dati globali
  - heap
  - parametri e variabili d’ambiente

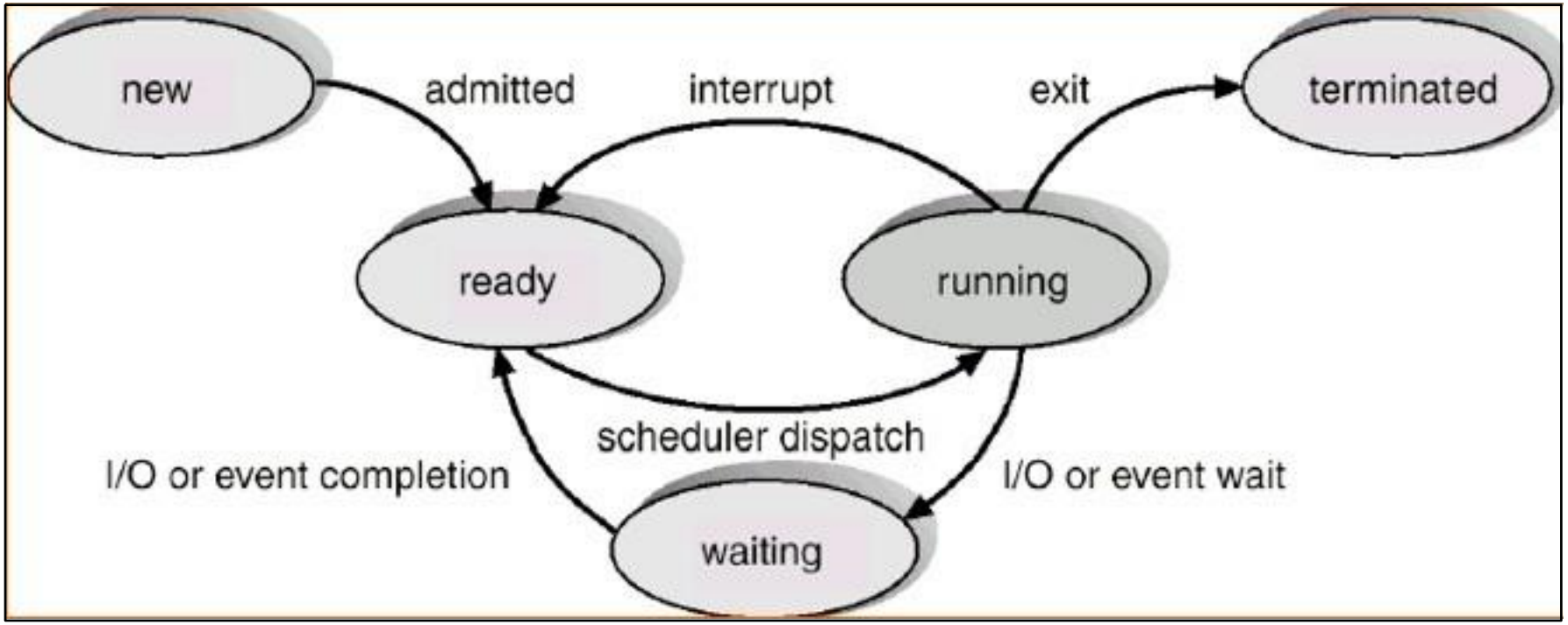
  Ovviamente, tutto ciò (tranne i registri) è contenuto in memoria

STATO DI UN PROCESSO

- Dal momento in cui entra in esecuzione, nel proprio ciclo di vita un processo può cambiare più volte stato
  - lo stato è determinato dal *tipo di attività* che il processo sta svolgendo (es. eseguendo istruzioni CPU, in attesa di operazioni I/O, etc)

- In generale i possibili stati di un processo sono:
  - **new**
    - il processo è appena stato creato
  - **running**
    - il processo sta eseguendo istruzioni
  - **waiting**
    - il processo è in attesa di un evento (es. completamento operazione di I/O o ricezione di un segnale da un altro processo)
  - **ready**
    - pronto ad eseguire istruzioni, in attesa di ricevere un processore
  - **terminated**
    - il processo ha completato l'esecuzione

DIAGRAMMA DEGLI STATI DI UN PROCESSO



PROCESSI:
HW TASK SWITCH vs. SW TASK SWITCH

- Abbiamo visto che i processori della famiglia IA-32 (come tante altre famiglie) mettono a disposizione del sistema operativo istruzioni privilegiate e registri specializzati, per manipolare strutture dati in memoria che mantengono lo stato dei task in esecuzione sul computer. In particolare mantengono strutture di tipo Task State Segment ed hanno un registro che punta al task corrente.
- Queste caratteristiche consentirebbero al sistema operativo di realizzare uno **switch tra i task assistito dall'hardware**, e quindi assai efficiente. In tal modo, aumenterebbe l'efficienza dei meccanismi di scheduling dei processi.

- Il limite di questo approccio è dovuto al fatto che, processori di famiglie diverse offrono istruzioni, registri e strutture dati **diverse** per sostenere lo switch hardware di task. Inoltre, alcuni processori, ad esempio quelli della famiglia AMD64 **non offrono** affatto questi servizi.

- Quindi, il sistema operativo, **dovendo operare su architetture diverse**, dovrebbe **assorbire le differenze** tra le diverse architetture, **implementando completamente in software parti più o meno cospicue di un meccanismo di switch tra task**.

PROCESSI: SW TASK SWITCH

- La scelta operata da molti sistemi operativi è stata perciò quella di **abbandonare lo switch hardware** di task, offerto dai processori, e di **realizzare lo switch di task principalmente in software**.
- Fino a Windows 3.1 si usava lo switch HW. Da Windows NT 3.1 ha scelto lo switch SW.

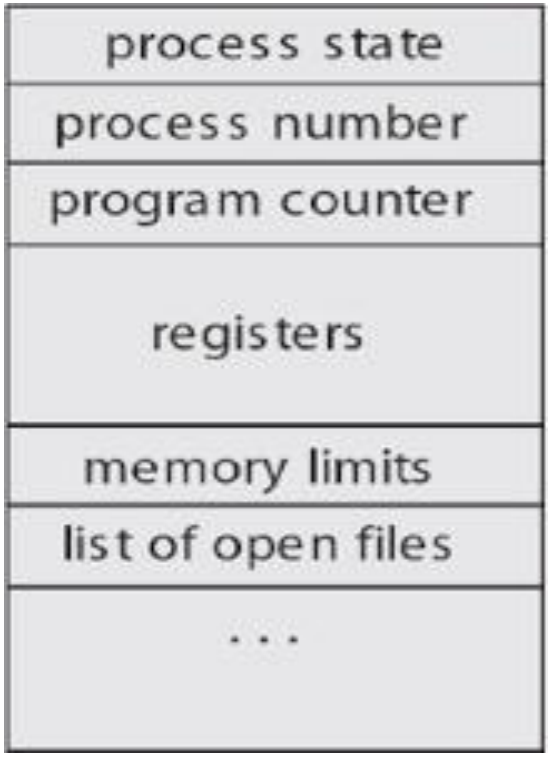
- Da questa scelta dei sistemi operativi è successivamente derivata la scelta delle famiglie più recenti di processori di abbandonare essi stessi il supporto allo switch hardware dei task. Attualmente, sia AMD64 che x86-64 non offrono un supporto completo allo switch hardware dei task.
- Viene però **ancora fornito dai processori un supporto hardware** per la gestione di alcuni caratteristiche importanti dei task, ad esempio la gestione degli **stack dedicati ai diversi livelli di esecuzione** (ring) della CPU.

- I sistemi operativi recenti, quindi, **pur non ricorrendo più allo switch hardware dei task, utilizzano ancora alcune funzionalità hardware relative ai task.**
- Anche Linux e Windows implementano lo switch software.

- Se il processore richiede obbligatoriamente la definizione dei task e la presenza della struttura TSS per ciascun task, allora **i sistemi operativi implementano un unico task e lo utilizzano, senza switch hardware, per tutti i processi**, **e implementano poi lo switch software tra i processi**.

PROCESS CONTROL BLOCK (PCB)

- Un sistema operativo ha strutture dati apposite con cui tiene traccia di tutti i processi in esecuzione e del loro stato.
- In particolare per ogni processo viene creato un **Process Control Block** (*PCB*) o *task control block* ove si mantengono informazioni specifiche sul processo e il suo stato.
- Questo PCB, nella gestione dei processi, svolge un ruolo analogo al ruolo del Task State Segment nella gestione dello switch HW dei Task per l'arch. IA-32.
- In particolare un PCB contiene:
  - id del processo, del processo genitore
  - stato del processo (attivo, sospeso, ..)
  - contesto
    - program counter
    - registri del processore
  - informazioni sullo scheduling del processore
  - informazioni sulla gestione della memoria
  - informazioni di accounting
  - informazioni sulle operazioni di I/O



PCB IN SISTEMI UNIX

- Informazioni contenute nel PCB in sistemi UNIX
  - PID del processo, PID del processo padre (PPID), ID utente (user ID)
  - stato
  - descrittore di evento
    - per un processo in stato di sleeping, indica l'evento per cui il processo è in attesa
  - puntatori alle tabelle delle pagine
  - dimensione delle tabelle delle pagine
  - posizione della user area
  - priorità
  - segnali pendenti

IMPLEMENTAZIONE DI UN PCB: LINUX

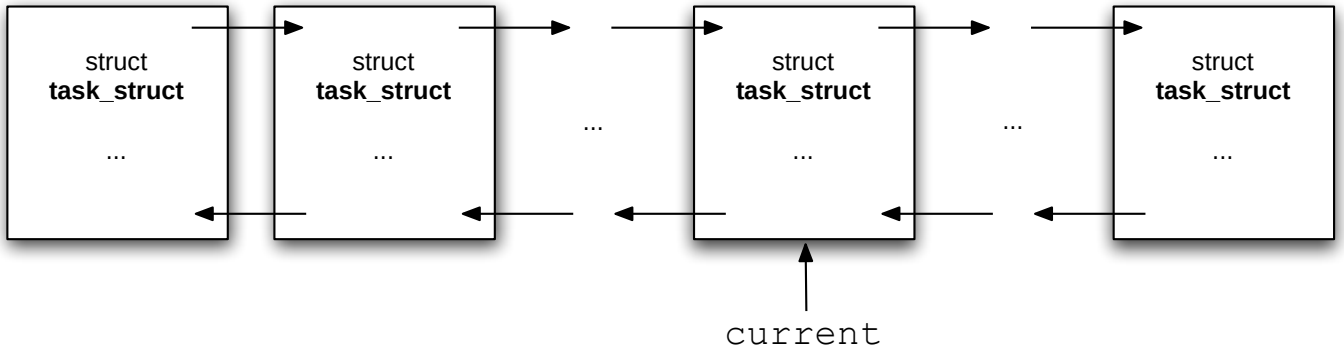
- E' rappresentato da una struttura **`task_struct`** molto corposa
  - dichiarata in `./include/linux/sched.h`
- Alcuni campi:

```
struct task_struct {
  volatile long state;     /* -1 unrunnable, 0 runnable, >0 stopped */
  void *stack;
  ...
  pid_t pid;    /* process identifier                        */
  ...
  struct task_struct *parent;   /* processo genitore del processo        */
  struct list_head children;    /* lista dei figli del processo          */
  struct files_struct *files;   /* lista dei file aperti                 */
  struct mm_struct *mm;         /* memoria del processo */
  ...

}
```

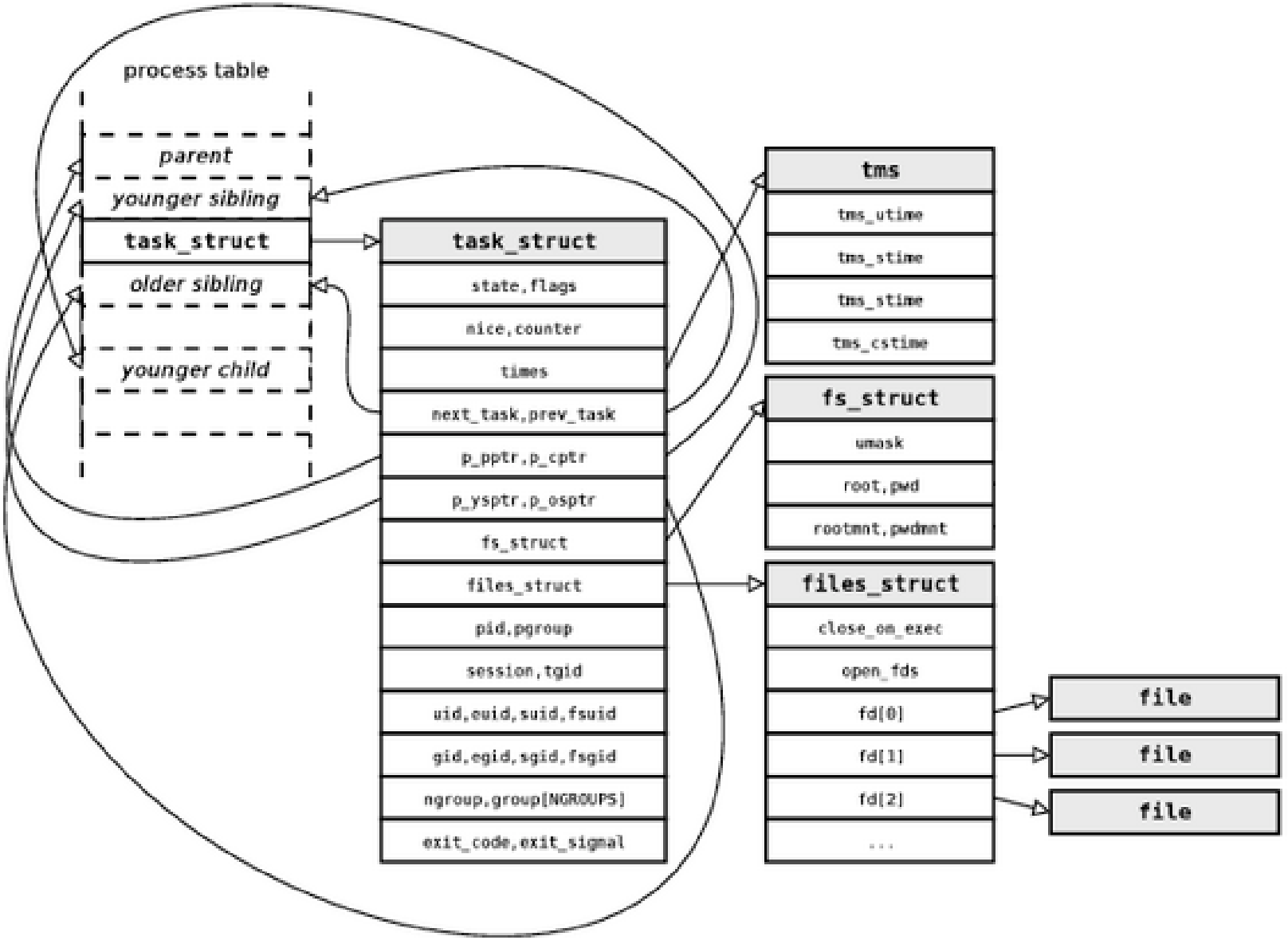
TABELLA DEI PROCESSI

- Struttura dati con cui il kernel tiene traccia di tutti i processi attivi nel sistema
  - ogni entry della tabella è il PCB di un processo attivo
  - tipicamente ha una dimensione massima prefissata
    - limite massimo al numero di processi che possono essere creati
- Esempio Linux
  - la tabella dei processi è implementata come lista doppiamente concatenata



  - Il kernel mantiene aggiornato un puntatore **current,** che punta al descrittore del processo correntemente in esecuzione

TABELLA DEI PROCESSI IN LINUX



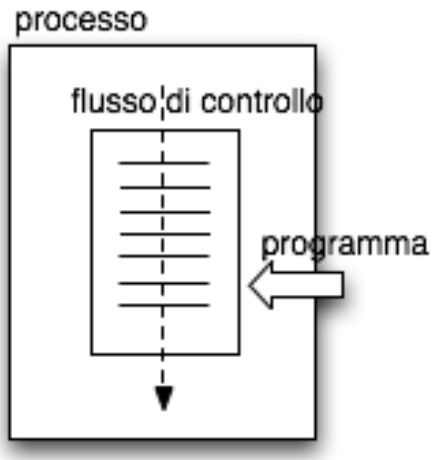
USER AREA IN PROCESSI UNIX/LINUX

- Nei sistemi Unix**, le informazioni di sistema relative ad un processo** sono mantenute in parte **nella tabella dei processi**, in parte **in un'area di memoria, appartenente al processo stesso e chiamata user area**

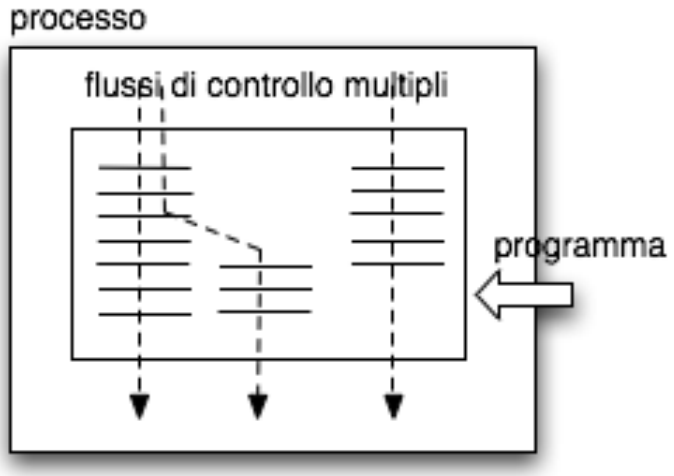
  - è localizzata al termine della parte superiore dello spazio di indirizzamento del processo
  - è accessibile solo quando il sistema esegue in kernel mode
  - contiene
    - puntatore all'ingresso relativo al processo nella process table
    - setting dei segnali
    - tabella dei file del processo
    - directory di lavoro corrente
    - umask settings
- La user area può essere sottoposta a swap su disco.
  - al contrario la tabella dei processi, che non può essere swappata

PROCESSI E THREADS

- Ad un processo sono associati uno o più flussi di controllo definiti **threads**
  - per flusso di controllo si intende l'esecuzione sequenziale di istruzioni da parte dell'esecutore del processo
- Nei moderni sistemi operativi, un processo può avere più flussi di controllo mediante *multi-threading*



processo con
singolo flusso di controllo



processo con
più flussi di controllo (threads)

CHIAMATE DI SISTEMA RELATIVE AI PROCESSI

- I sistemi operativi mettono a disposizione servizi - sempre forniti in forma di system call - per varie operazioni che concernono i processi, in particolare:
  - recupero informazioni di stato
  - creazione
  - terminazione
  - comunicazione
  - sincronizzazione / coordinazione
  - (mobilità)

RECUPERO INFORMAZIONI

- Esistono varie funzioni per recuperare informazioni sui processi
  - ad esempio l'ID dei processi
- Nello standard POSIX ad esempio esiste la chiamata **getpid** con cui è possibile ottenere l'ID di un processo:

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

int main (void){
  printf("hello world from process ID %d \n", getpid());
  exit(0);
}
```

  - lanciando più volte il programma (dopo averlo compilato) si ottiene un output del tipo:

hello world from process ID 10590

hello world from process ID 10591

CREAZIONE DI PROCESSI

- E’ tipicamente definita in modo gerarchico
  - la creazione di un processo avviene ad opera del processo definito genitore *(parent)* e i processi generati vengono definiti figli *(children)*
  - i processi figli possono a loro volta creare processi, divenendo loro stessi processi parent
  - si creano quindi degli alberi di processi *(process tree)*

- La creazione di un processo implica l'allocazione di un certo insieme di risorse (memoria in primis) allo scopo
- Processi genitori e figli possono condividere o meno - a seconda dei vari S.O. - risorse
- In particolare nei sistemi operativi moderni ogni processo *ha il proprio spazio di indirizzamento di memoria*
  - **non c’è condivisione di memoria fra processi.**
  - per condividere memoria è necessario sfruttare opportuni meccanismi e funzioni forniti dal sistema operativo
    - shared memory, descritta in seguito, vedi mmap

CREAZIONE DI UN PROCESSO: `fork`

- Nei sistemi UNIX (POSIX) **fork** è la chiamata di sistema per creare un nuovo processo

```
pid_t          fork(void);
```

- L'esecuzione della chiamata comporta la creazione un nuovo processo, con uno spazio di memoria (dati, stack) che è *copia esatta*, al momento della creazione, di quello del padre
  - quindi un ulteriore flusso di controllo (del processo figlio)

- A livello di programma, il processo figlio parte ad eseguire direttamente alla fine della chiamata della fork
  - quindi sia il padre, sia il figlio continuano l'esecuzione dal punto in cui è stata terminata la chiamata alla fork

- Per distinguere padre dal figlio:
  - nel caso del processo figlio, la fork restituisce come valore di ritorno 0
  - nel caso del processo padre, la fork restituisce un numero intero pari all'*identificatore di processo del figlio*

ESEMPIO CREAZIONE PROCESSO

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

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

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

    if (pid<0){
        printf("Fork failed.\n");
        exit(1);
    } else if (pid==0){ /* processo figlio */
        printf("[NEWPROC] started \n");
        sleep(5);
        printf("[NEWPROC] completed \n");
    } else { /* processo genitore */
        printf("[PARENT] waiting for child proc %d\n",pid);
        wait(NULL);
        printf("[PARENT] child proc completed. \n");
        exit(0);
    }
}
```

-il processo padre crea un processo figlio con **fork**

-Il processo figlio aspetta 5 secondi, poi termina

-Il padre aspetta la terminazione del figlio (con **wait**) e poi esce

ATTESA TERMINAZIONE DEI FIGLI

- **wait**
  - sospende l'esecuzione del processo padre fino a quando *uno qualsiasi dei processi figli* ha terminato la propria esecuzione

```
#include <sys/wait.h>

pid_t wait(int *stat_loc);
```

una funzione più potente permette di attendere la terminazione di uno specifico figlio o di un figlio qualunque, oppure di continuare se nessun figlio è terminato.

- **waitpid**:
  - sospende l'esecuzione del processo padre fino alla terminazione di uno specifico processo figlio

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

ESEMPIO ATTESA TERMINAZIONE FIGLI

- Processo padre che crea N figli
  - ogni figlio stampa in standard output il proprio PID e attende una certo numero (casuale) di millisecondi prima di terminare
  - il padre aspetta la terminazione di tutti i figli e poi termina

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

int getrand(int base, int range){
  int c = clock(); srand(c); return base + rand()%range;
}

void proc(){
  pid_t mypid;  int dt;

  mypid = getpid();
  dt = getrand(500,1500);
  printf("Process %d started - waiting for %d ms.\n",mypid,dt);
  usleep(dt*1000);
  printf("Process %d completed.\n",mypid);  exit(0);
}

int main(int argc, char** argv){
  pid_t pid; int i; int nprocs;
  nprocs = atoi(argv[1]);
  printf("[PARENT] Spawning %d processes \n",nprocs);  for (i = 0; i < nprocs;
  i++){
    pid = fork();
    if (pid < 0) {
         printf("Error in process creation (%d)\n",i);
    } else if (pid == 0){
         proc();
    }
  }
  printf("[PARENT] Waiting for child processed...\n");
  for (i = 0; i < nprocs; i++){ wait(NULL); }
  printf("[PARENT] All processes terminated.\n");
  exit(0);
}
```

ISOLAMENTO

- I processi *non* condividono memoria
  - nei sistemi UNIX alla creazione con fork il processo figlio ha una *copia* dell'immagine completa di memoria del processo padre
  - Va però considerato che, mentre i file descriptor e le tabelle dei file aperti di un processo, appartengono all'immagine del processo e perciò vengono duplicati, la tabella dei file aperti nel sistema è unica e non viene duplicata con la fork. Vedere slide successiva.

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

int value = 0;

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

  pid = fork();
  if (pid == 0){
    /* processo figlio */
    value++;
    exit(0);
  } else if (pid > 0){
    /* padre */
    /* aspetta terminazione del figlio */
    wait(NULL);
    printf("%d\n",value);
    exit(0);
  }
}
```

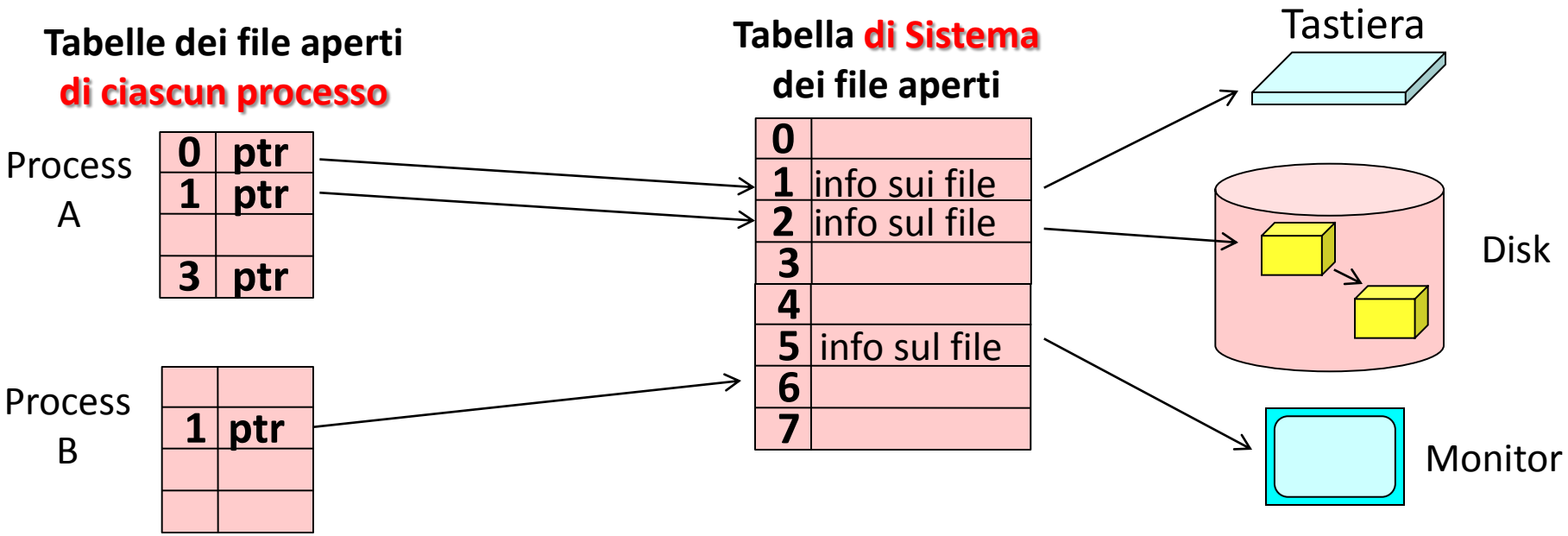
In output viene stampato (dal padre) il valore 0

- se ci fosse stata condivisione di memoria, padre e figlio avrebbero condiviso la variabile value e il valore in output sarebbe stato 1, dopo l'incremento del figlio

File Descriptors e Tabelle dei File Aperti

Il **file descriptor** è un'astrazione per permettere l'accesso ai file. Ogni file descriptor e' rappresentato da un integer.

Ogni processo ha la propria **file descriptor table** che contiene (indirettamente) le informazioni sui file attualmente utilizzati (aperti) dal processo stesso. In particolare, la file descriptor table contiene un file descriptor (un intero) per ciascun file utilizzato dal processo.
Per ciascuno di questi file descriptor, la tabella del processo punta ad una tabella di sistema che contiene le informazioni sui tutti i file attualmente aperti dal processo stesso.
Si noti che due diversi processi possono avere due diversi file descriptor con stesso valore, ma che fanno riferimento a file diversi



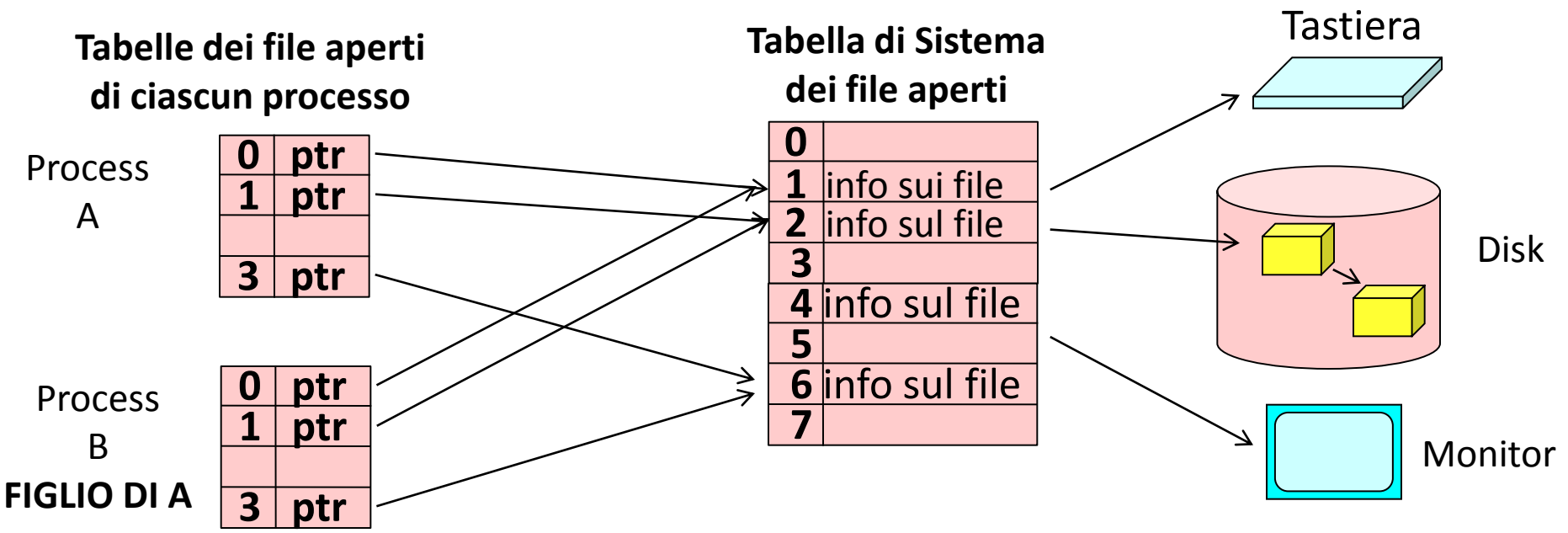
In POSIX ogni processo è inizialmente associato a 3 fd standard
stdin (0), stdout (1), stderr (2)

Duplicazione delle sole tabelle dei file descriptor dei singoli processi

Duplicando il processo, il figlio ha una tabella dei file aperti nel processo che è una copia esatta della tabella del padre.

Invece, la tabella di sistema non viene duplicata.
Quindi più processi (padre e figli) accedono, in lettura e scrittura, alle informazioni contenute nelle entry comuni nella tabella di sistema dei file aperti.

Nel momento della duplicazione di un processo, vengono incrementati di 1 i contatori dei file che erano aperti dal padre. Man mano che padre e figlio chiudono i file, questi contatori vengono decrementati di 1. La chiusura effettiva di un file avviene solo quando il contatore per quel file diventa zero.



ESECUZIONE DI UN PROGRAMMA ESTERNO: `exec`

- Nei sistemi UNIX (POSIX) la famiglia di chiamate di sistema **exec** permettono di caricare e mandare in esecuzione un programma eseguibile

```
#include <unistd.h>
extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ..., char * const envp[]);
int execv(const char *path, char *const  argv[]);
int execvp(const char *file, char *const argv[]);
```

  - **execl**
    - carica in memoria ed esegue un programma, nel contesto del processo corrente
    - varianti: execv, execle, execlp, execvp
      - a seconda che siano specificati anche parametri al programma da eseguire e le variabili d'ambiente

CREZIONE PROCESSO ED ESECUZIONE COMANDO SHELL

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


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

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

    if (pid<0){
        fprintf(stderr,"Fork fallita.\n");
        exit(1);
    } else if (pid==0){ /* processo figlio */
        execl("/bin/ls","");
    } else { /* processo genitore */
        wait(NULL);
        printf("Processo figlio terminato.\n");
        exit(0);
    }
}
```

-il processo padre crea un processo figlio con **fork**

-Il processo figlio carica ed esegue con **execlp** il programma di sistema ‘ls’ (che ha come effetto il listing della directory corrente)

-Il padre aspetta la terminazione del fliglio (con **wait**) e poi esce

TERMINAZIONE DI UN PROCESSO

- La terminazione di un processo può avvenire in varie circostanze:
  - quando viene eseguito l'ultima istruzione del suo programma e/o viene effettuata una chiamata di sistema apposita, chiamata **exit**
    - specificando un **exit status**, che può essere letta dal processo genitore (mediante wait) e che contiene informazioni circa la corretta terminazione o meno del processo
  - quando un processo esterno (tipicamente il genitore) ne richiede la terminazione, mediante system call **abort**

- In alcuni sistemi la terminazione di un processo implica automaticamente anche la terminazione di tutti i figli (**cascading termination**)

- Nei sistemi UNIX **non c' è cascading termination**
  - può quindi capitare che un processo termini prima che i propri figli abbiano completato la propria esecuzione
  - in tal caso un processo figlio si dice **orfano** e viene automaticamente "adottato" dal processo **init** (che ha PID = 1)

PROCESSI ZOMBIE

- Quando un processo termina, tutta la memoria e le risorse associate vengono liberate
- Tuttavia, nei sistemi UNIX l'entry nella tabella dei processi rimane in modo tale che il processo genitore del processo terminato possa leggere l'exit status mediante una **wait**

- Fintanto che il processo padre non esegue la wait, il processo figlio viene definito **zombie**
  - è terminato, ma la sua entry c'è ancora...
- Eseguendo la wait, l'entry viene correttamente rimossa

- Quindi è fondamentale che un processo esegua la wait per ogni processo figlio che ha creato
  - per evitare di riempire la tabella dei processi con entry di processi già terminati

- Il problema degli zombie si risolve anche qualora il processo padre muoia senza avere fatto la wait per i suoi figli. Infatti, i figli zombie, morto il padre diventano zombie e **orfani**, vengono adottati dal processo **init**, che ogni tanto fa le **wait** per i propri figli adottati e così fa rilasciare le risorse.