Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative
Istituto sistemi informativi e networking

SUPSI

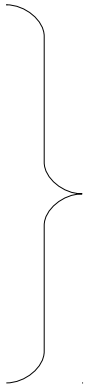
Programmazione multiprocesso

Amos Brocco, Ricercatore, DTI / ISIN

20 febbraio 2012

Processi

- Cos'è un **processo**?
  - Un programma in esecuzione
    - Spazio di memoria
      - Codice macchina (text)
      - Dati
      - Stack
      - Heap
    - Puntatori a
    - Registri
    - Program Counter
    - Descrittori di file
    - PID
    - Stato (in esecuzione, bloccato,...)
    - ...

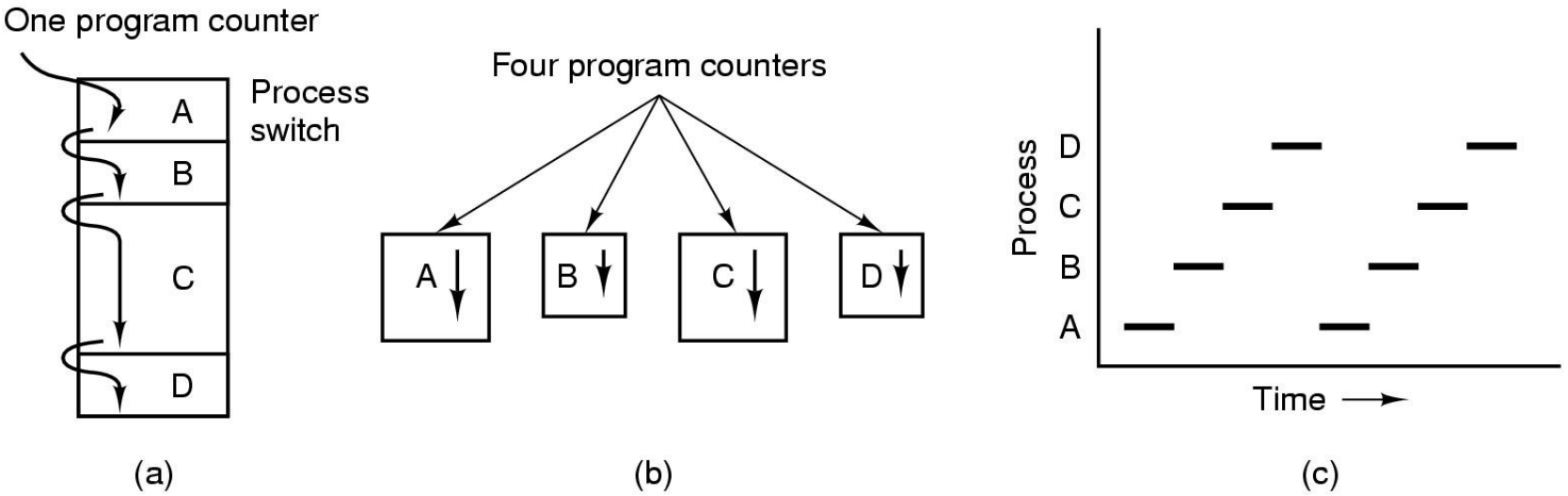


Descrittore di processo (**Process Control Block**, PCB)

Sistemi multiprogrammati

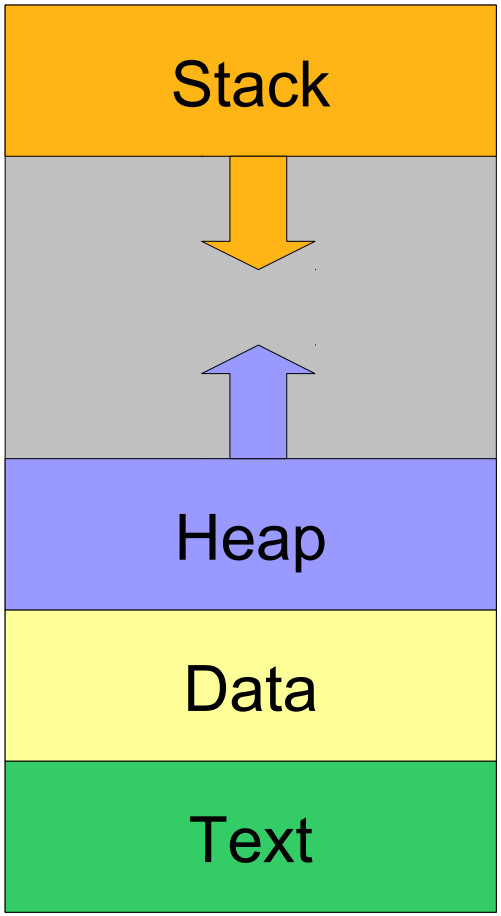
- Sistemi **multiprogrammati**
  - **più processi contemporaneamente in memoria**
  - Il kernel mantiene una tabella dei processi (*process table*) con le informazioni necessarie all'esecuzione di ogni processo (*process control block*)
  - Esecuzione:
    - Pseudo parallelismo (time sharing, CPU singola)
    - Vero parallelismo (multi-core, multi-processore)

Sistemi multiprogrammati



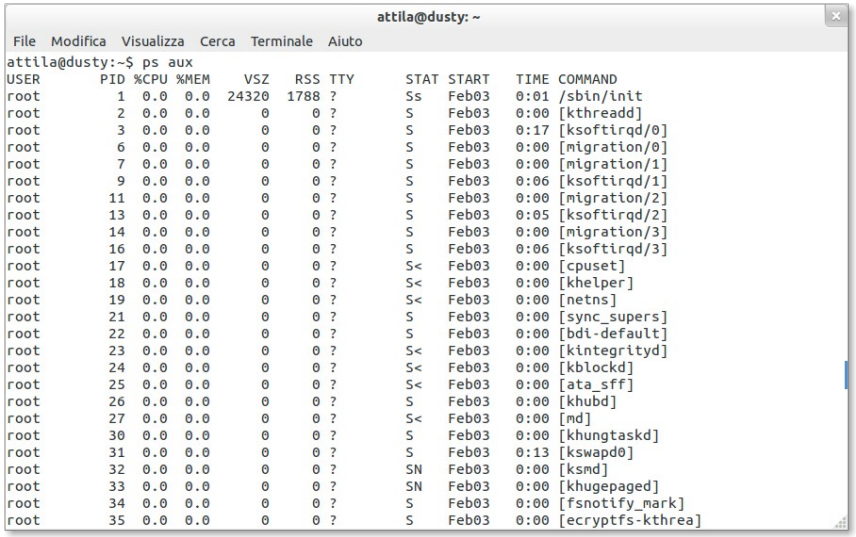
A. Tanenbaum, Modern Operating Systems, 2nd ed.

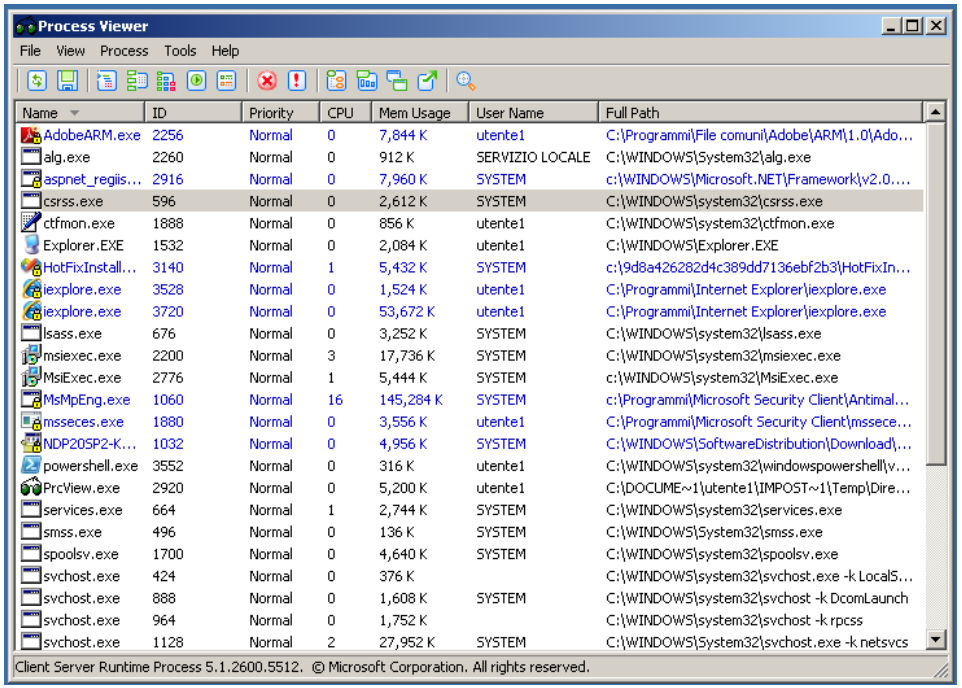
Spazio di indirizzamento di un processo



Organizzazione dei processi

- Il sistema operativo (kernel) identifica univocamente ogni processo tramite un **PID** (process identifier)
- Il **PID** è assegnato alla creazione del processo





getpid

- La funzione **getpid** permette di ottenere il pid assegnato al processo corrente

```
#include <unistd.h>

pid_t getpid(void);
```

Creare processi in Unix: fork

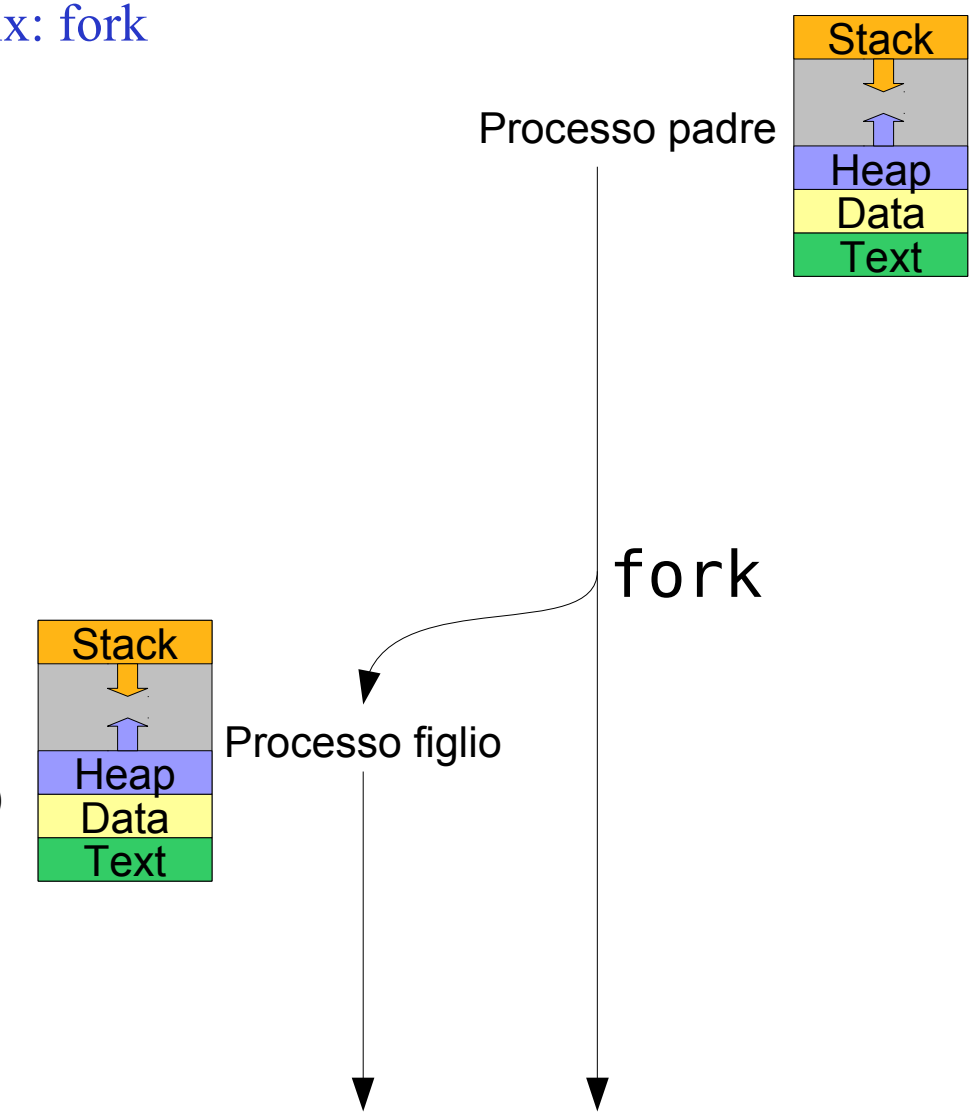
```
#include <unistd.h>

pid_t fork(void);
```

1. Crea un nuovo spazio privato in memoria per il figlio
2. Crea una nuovo descrittore di processo per il figlio (→**nuovo PID**)
3. Copia lo spazio di memoria del padre in quello del figlio
4. Valore di ritorno:

   Al padre: ritorna il PID del figlio
   Al figlio: ritorna 0

   Se il fork fallisce: ritorna -1

Creare processi in Unix: fork



Viene creato un nuovo spazio di indirizzamento (copia di quello del padre)

Esempio fork

```
#include <unistd.h>

pid_t cpid;

cpid = fork();

if (cpid == (pid_t) -1) {
      printf("Errore!\n");
} else if (cpid == 0) {
      printf("Sono il figlio\n");
} else {
      printf("Sono il padre\n");
}
```

Gerarchia di processi

- Con fork viene creata una gerarchia di processi
  - In cima all'albero c'è il processo **init** (PID=1)
- Un processo può determinare il PID del padre con **getppid** (get parent pid)

```
#include <unistd.h>

pid_t getppid(void);
```

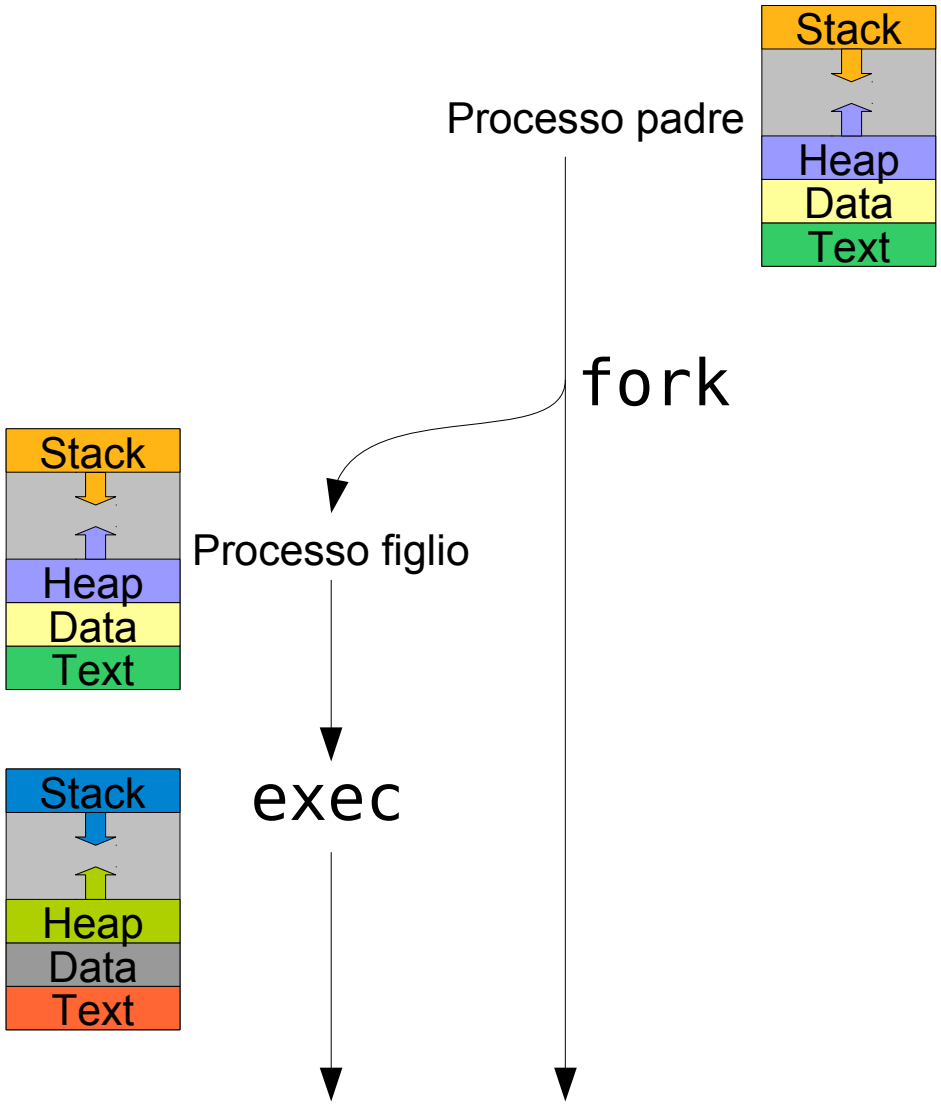
Eseguire un altro programma: Exec

```
#include <unistd.h>

int execl(const char *path, const char*arg0, ...);
int execlp(const char *file, const char *arg0, ...);
...
```

Sostituisce i segmenti Text e Data del processo con quanto caricato dal file del programma specificato

Esistono diverse varianti di exec (cf. `man exec`)

Exec



Viene creato un nuovo spazio di indirizzamento (copia di quello del padre)

I segmenti Text e Data vengono sostituiti

Terminare un processo

- Un processo termina quando
  - **`return`** viene invocato dalla procedura main
  - viene invocata la procedura **`exit(int)`**
  - Il valore di ritorno/uscita può essere letto dal processo padre

```
#include <stdlib.h>

void exit(int status);
```

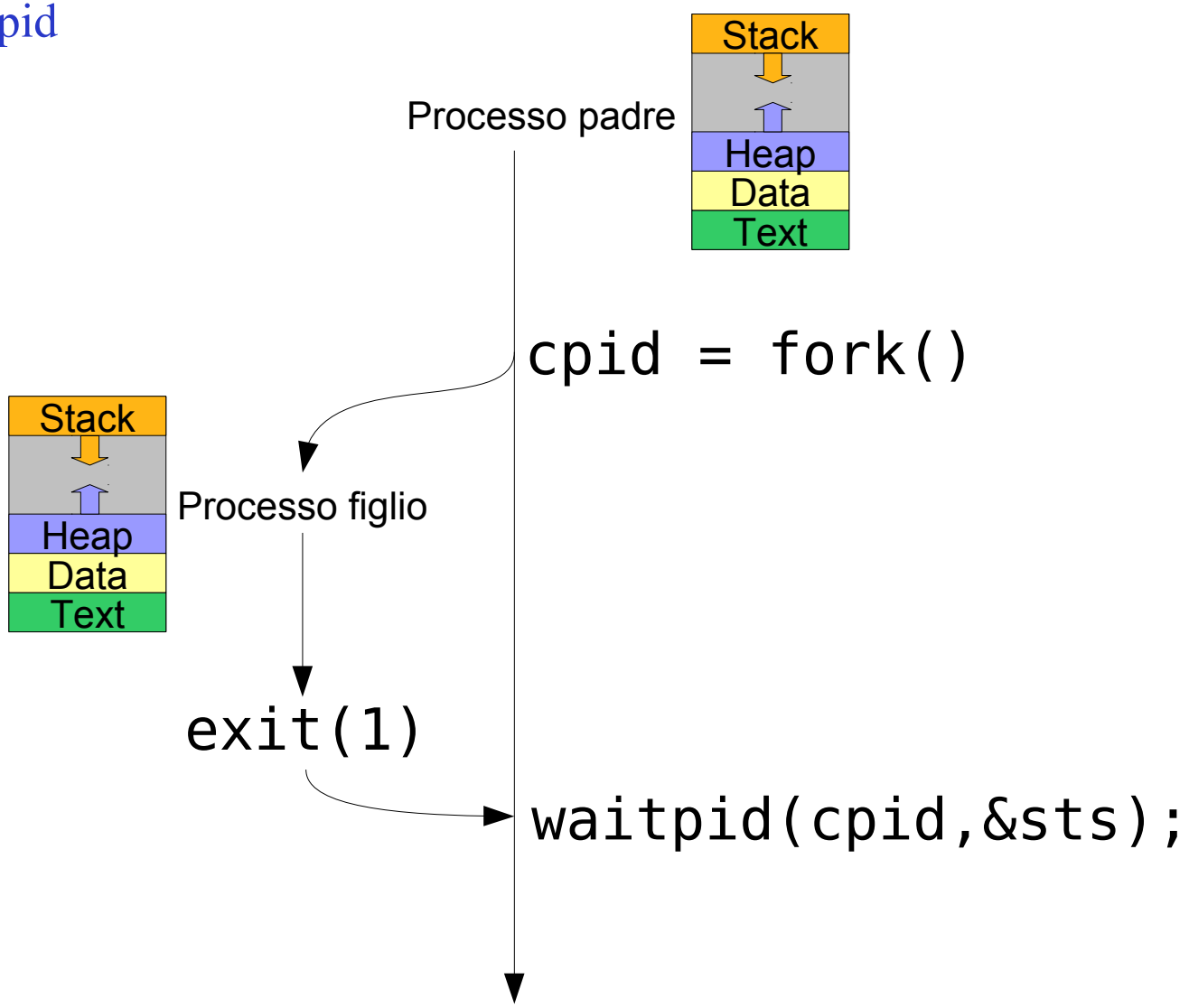
Attendere il valore di uscita di un processo figlio: Wait / Waitpid

- Il processo padre aspetta la terminazione dei processi figli e ottiene lo stato di uscita con le funzioni **wait** e **waitpid**

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

pid_t wait(int *status);

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

* La terminazione è segnalata con SIGCHLD

waitpid


Processo padre
Stack
Heap
Data
Text
cpid = fork()
Stack
Heap
Data
Text
Processo figlio
exit(1)
waitpid(cpid,&sts);

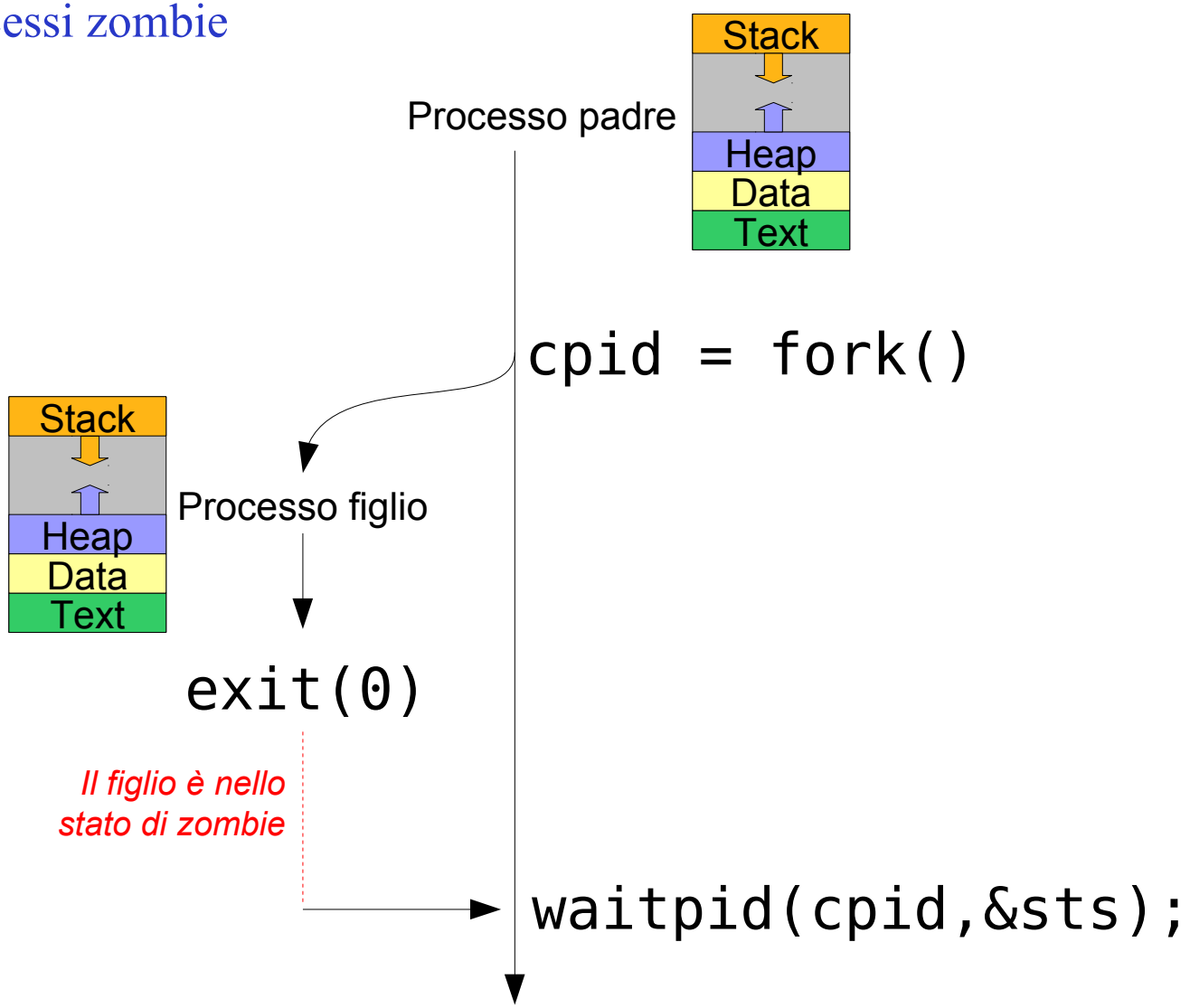
Esempio waitpid

```
#include <unistd.h>
#include <wait.h>
pid_t cpid;
int sts;
int main(void) {
  if (cpid = fork()) {
    waitpid(cpid, &sts, 0);
    printf("Figlio uscito con %d\n", WEXITSTATUS(sts));
  } else if (cpid == 0) {
    printf("Sono il figlio\n");
    exit(1);
  }
}
```

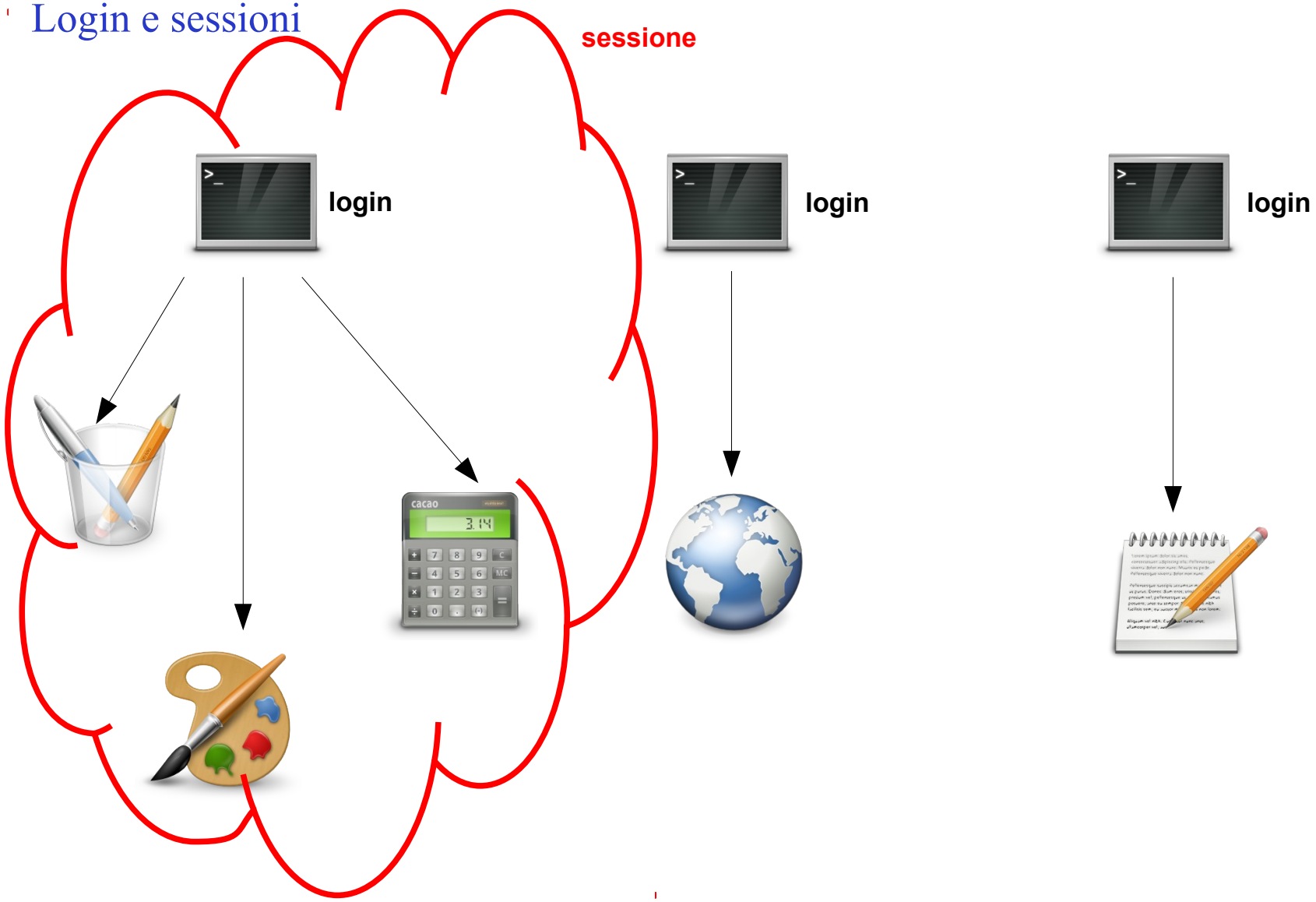
Processi zombie

- Quando il sotto-processo figlio termina è necessario che il padre sia in attesa con **`wait`** o **`waitpid`**
- Fintanto che il padre non attende, il figlio rimane nello stato **zombie** (o defunct) e non può essere rimosso dal sistema
  - Quando il padre termina il figlio viene rimosso
  - Se il padre termina prima del figlio, il sotto-processo viene ereditato da init

Processi zombie



Login e sessioni



Sessioni

- Quando un utente si collega al sistema il processo di login crea una nuova sessione
  - Ogni sessione ha un proprio identificatore (**SID**, **S**ession **ID**)
  - Tutti i sottoprocessi ereditano automaticamente l'identificatore di sessione

- Quando l'utente abbandona il sistema (logout) tutti i processi in esecuzione all'interno della sessione possono essere facilmente identificati e terminati

Sessioni

- Con **getsid** posso ottenere il SID

```
#include <unistd.h>

pid_t getsid(pid_t pid);
```

- Dopo un fork, un processo può creare una nuova sessione con **setsid**

```
#include <unistd.h>

pid_t setsid(void);
```

Gruppi di processi

- La filosofia Unix si basa sull'utilizzo di programmi semplici che possono essere combinati facilmente per svolgere operazioni complesse

```
ls *.txt | grep Hello | sort | head -n 5
```

Per controllare facilmente tutti i processi creati da questa linea di comando (p.es. per terminarli) è possibile raggrupparli...

Gruppi di processi

- Ogni processo è membro di un gruppo di processi
- Ogni gruppo è identificato univocamente da un **Process Group ID** (PGID)
- Ogni gruppo ha un leader, il cui PID è uguale al PGID del gruppo
- Ogni gruppo ha associato un terminale di controllo

Gruppi di processi

- Un processo eredita il gruppo dal padre, può conoscere il PGID con **`getpgid`**, e può cambiarlo con **`setpgid`**

```
#include <unistd.h>

pid_t getpgid(pid_t pid);

int setpgid(pid_t pid, pid_t pgid);
```

Se pgid è uguale pid o uguale a 0 viene creato un nuovo gruppo, di cui il processo è leader