LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

Nota sugli “snippet” di codice

Alcuni esempi di codice possono essere semplificati, ad esempio omettendo il blocco principale con la funzione `main` *(che andrebbe aggiunto) oppure elencando alcune o tutte le librerie da includere tutte su una riga o insieme (per cui invece occorre trascrivere correttamente le direttive* `#include` *secondo la sintassi corretta) o altre semplificazioni analoghe. In questi casi occorre sistemare il codice perché possa essere correttamente compilato e poi eseguito.*

Architettura

Kernel Unix

Il kernel è l'elemento di base di un sistema Unix-like, ovvero il nucleo del sistema operativo. Il kernel è incaricato della gestione delle risorse essenziali: CPU, memoria, periferiche, ecc...

Ad ogni boot il sistema verifica lo stato delle periferiche, monta la prima partizione (root file system) in read-only e carica il kernel in memoria. Il kernel lancia il primo programma (*systemd*, sostituto di *init*) che, a seconda della configurazione voluta (target), inizializza il sistema di conseguenza.

Il resto delle operazioni, tra cui l'interazione con l'utente, vengono gestite con i programmi eseguiti dal kernel.

Kernel e memoria virtuale

I programmi utilizzati dall'utente che vogliono accedere alle periferiche chiedono al kernel di farlo per loro.

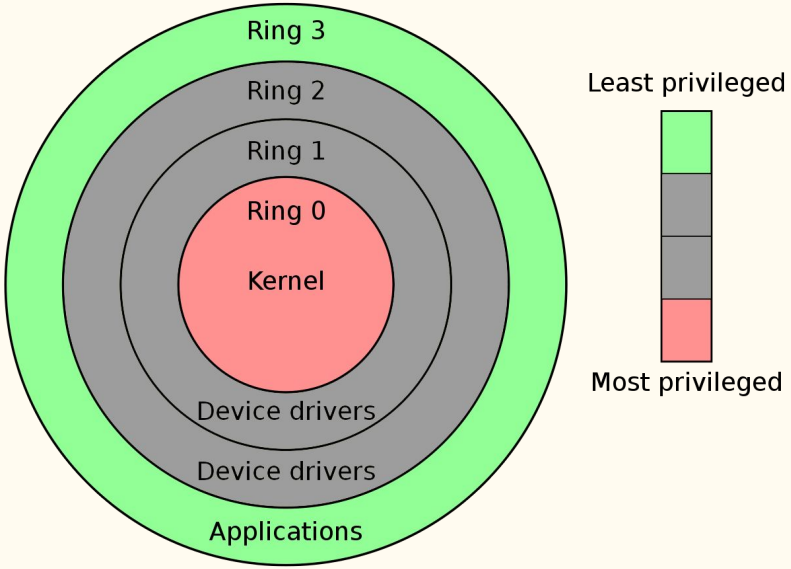
L'interazione tra programmi ed il resto del sistema viene mascherata da alcune caratteristiche intrinseche ai processori, come la gestione hardware della memoria virtuale (attraverso la MMU).

Ogni programma vede se stesso come **unico possessore della CPU** e non gli è dunque possibile disturbare l'azione degli altri programmi → stabilità dei sistemi Unix-like!

Privilegi

Nei sistemi Unix-like ci sono due livelli di privilegi:

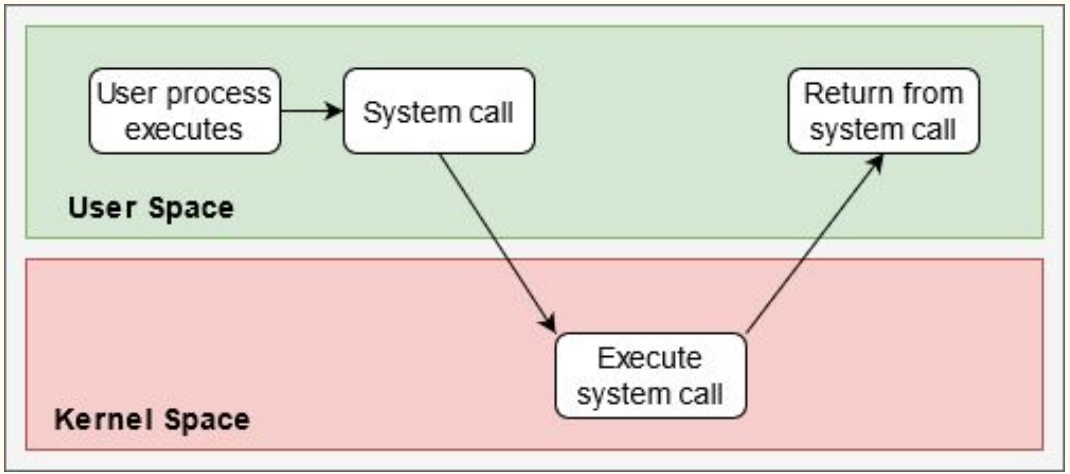
- **User space:** ambiente in cui vengono eseguiti i programmi.
- **Kernel space:** ambiente in cui viene eseguito il kernel.



System calls

System calls

Le interfacce con cui i programmi accedono all'hardware si chiamano **system calls.** Letteralmente "chiamate al sistema" che il kernel esegue nel kernel space, restituendo i risultati al programma chiamante nello user space.



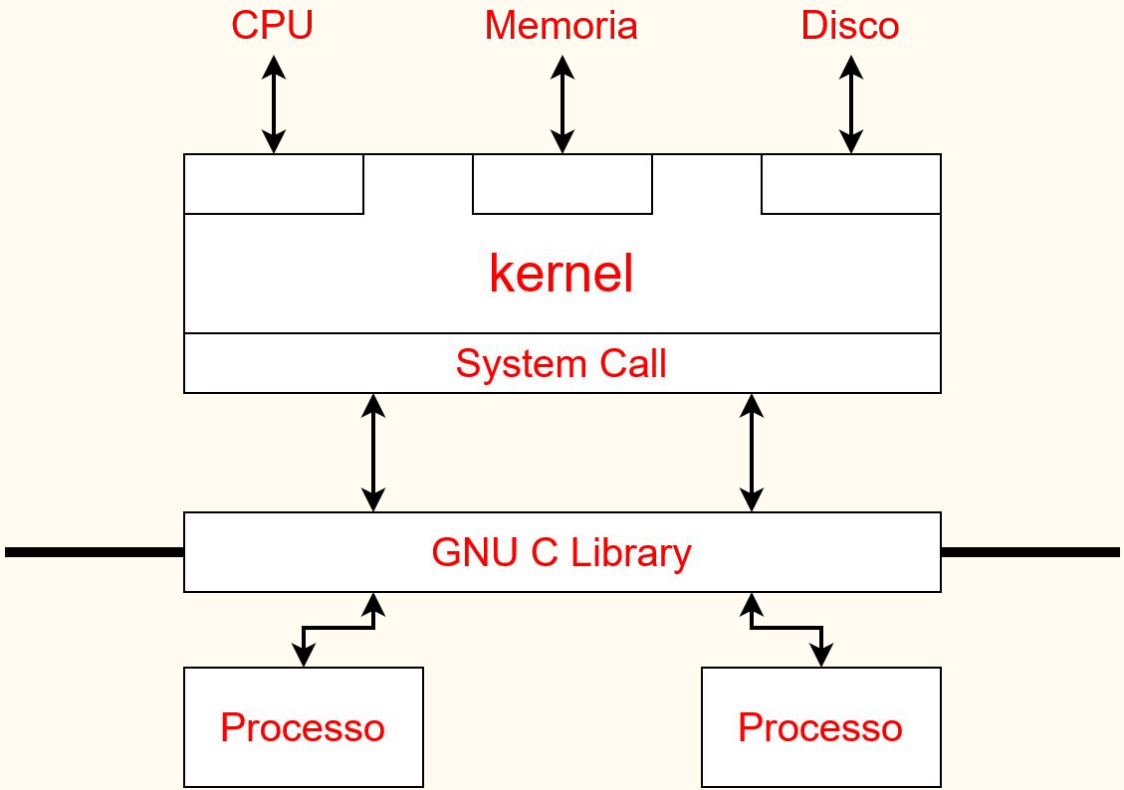
Le chiamate restituiscono "-1" in caso di errore e settano la variabile globale `errno`. Errori validi sono numeri positivi e seguono lo standard POSIX, il quale definisce degli alias.

System calls: librerie di sistema

Utilizzando il comando di shell `ldd` su di un eseguibile si possono visualizzare le librerie condivise caricate e, fra queste, vi sono tipicamente anche *ld-linux.so*, e *libc.so*.

- **ld-linux.so**: quando un programma è caricato in memoria, il sistema operativo passa il controllo a *ld-linux.so* anzichè al normale punto di ingresso dell'applicazione. ld-linux trova e carica le librerie richieste, prepara il programma e poi gli passa il controllo.
- **libc.so**: la libreria GNU C solitamente nota come glibc che contiene le funzioni basilari più comuni.

System Calls: librerie di sistema



Get time: time() e ctime()

```
time_t time( time_t *second )
char * ctime( const time_t *timeSeconds )
```

```c
#include <time.h>                                          //time.c
#include <stdio.h>

void main(){
    time_t theTime;
    time_t whatTime = time(&theTime); //seconds since 1/1/1970
    //Print date in Www Mmm dd hh:mm:ss yyyy
    printf("Current time = %s= %d\n", ctime(&whatTime),theTime);
}
```

Working directory: chdir(), getcwd()

```
int chdir( const char *path );
char * getcwd( char *buf, size_t sizeBuf );
```

```
#include <unistd.h>                                          //chdir.c
#include <stdio.h>

void main(){
    char s[100];
    getcwd(s,100); // copy path in buffer
    printf("%s\n", s); //Print current working dir
    chdir(".."); //Change working dir
    printf("%s\n", getcwd(NULL,100)); // Allocates buffer
}
```

Operazioni con i file

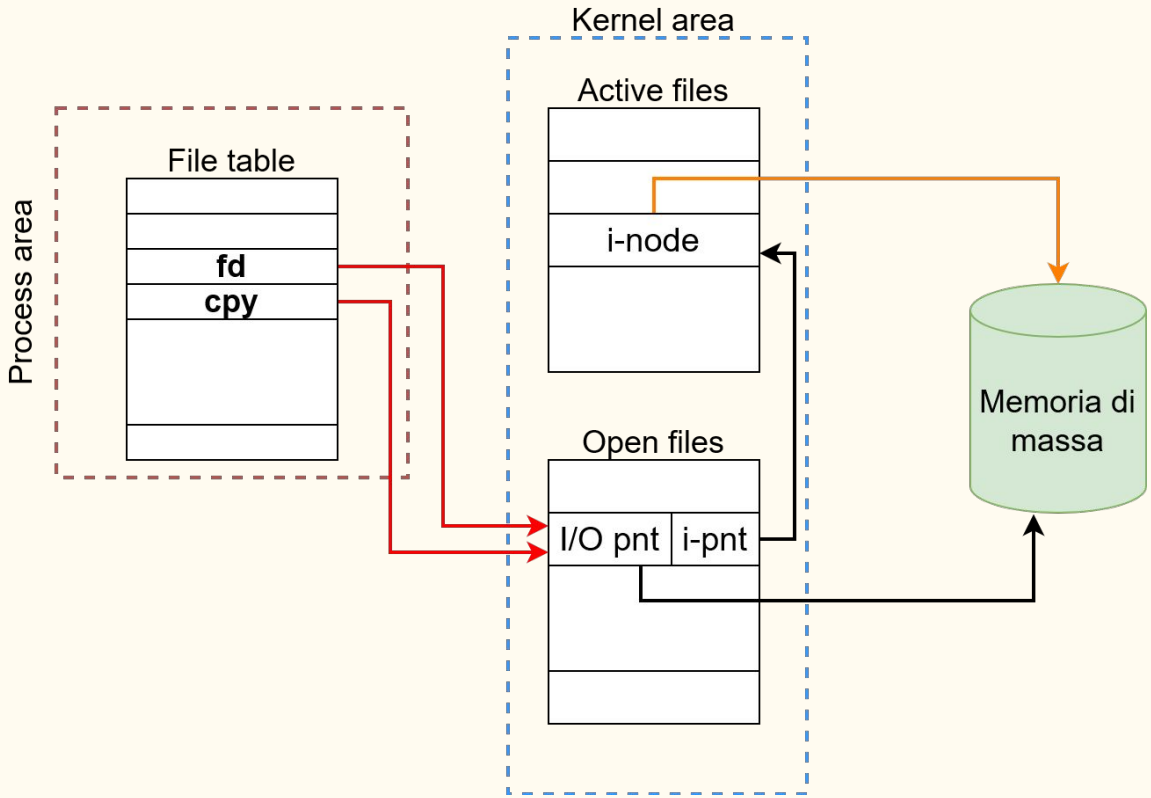
```
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);


FILE *fopen(const char *filename, const char *mode)
int fclose(FILE *stream)

.....
```

Duplicazione file descriptors: dup(), dup2()

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

```
#include <unistd.h> <stdio.h> <fcntl.h>
int main(void){
    char buf[51];
    int fd = open("file.txt",O_RDWR); //file exists
    int r = read(fd,buf,50); //Read 50 bytes from ‘fd’ in ‘buf’
    buf[r] = 0; printf("Content: %s\n",buf);
    int cpy = dup(fd); // Create copy of file descriptor
    dup2(cpy,22); // Copy cpy to descriptor 22 (close 22 if opened)
    lseek(cpy,0,SEEK_SET); // Move I/O on all 3 file descriptors!
    write(22,"This is a fine\n",16); // Write starting from 0-pos
    close(cpy); //Close ONE file descriptor
}
```

Duplicazione file descriptors: dup(), dup2()



Permessi: chmod(), chown()

```
int chown(const char *pathname, uid_t owner, gid_t group)
int fchown(int fd, uid_t owner, gid_t group)
int chmod(const char *pathname, mode_t mode)
int fchmod(int fd, mode_t mode)
```

//chown.c

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
void main(){
    int fd = open("file",O_RDONLY);
    fchown(fd, 1000, 1000); // Change owner to user:group 1000:1000
    chmod("file",S_IRUSR|S_IRGRP|S_IROTH); // Permission to r/r/r
}
```

```
$ cd .. ; touch file ; ls -la
$ ./executable.o ; ls -la
```

Eseguire programmi: execve() ed alias

```
int execv(const char *path, char *const argv[])
int execvp(const char *file, char *const argv[])
int execvpe(const char *file, char *const argv[],char *const
            envp[])

int execl(const char *path, const char * arg0, …,argn,NULL)
int execlp(const char *file, const char * arg0, …,argn,NULL)
int execle(const char *file, const char * arg0, …,argn,
           NULL, char *const envp[])

int execve(const char *filename, char *const argv[], char
           *const envp[])
```

Esempio: execv()

```c
#include <unistd.h>                                          //execv1.out
#include <stdio.h>
void main(){
    char * argv[] = {"par1","par2",NULL};
    execv("./execv2.out",argv); //Replace current process
    printf("This is execv1\n");
}
```

```c
#include <stdio.h>                                           //execv2.out
void main(int argc, char ** argv){
    printf("This is execv2 with %s and %s\n",argv[0],argv[1]);
}
```

Esempio: execle()

```c
#include <unistd.h>                                          //execle1.out
#include <stdio.h>
void main(){
    char * env[] = {"CIAO=hello world",NULL};
    execle("./execle2.out","par1","par2",NULL,env); //Replace proc.
    printf("This is execle1\n");
}
```

```c
#include <stdio.h>                                           //execle2.out
#include <stdlib.h>
void main(int argc, char ** argv){
    printf("This is execv2 with par: %s and %s. CIAO =
            %s\n",argv[0],argv[1],getenv("CIAO"));
}
```

Esempio: dup2/exec

```c
#include <stdio.h>                                          //execvpDup.c
#include <fcntl.h>
#include <unistd.h>

void main() {
  int outfile = open("/tmp/out.txt",
    O_RDWR | O_CREAT, S_IRUSR | S_IWUSR
  );
  dup2(outfile, 1); // copy outfile to FD 1
  char *argv[]={"./time.out",NULL}; // time.out della slide#10
  execvp(argv[0],argv); // Replace current process
}
```

Chiamare la shell: system()

```
int system(const char * string);
```

```
#include <stdlib.h> <stdio.h>                                    //system.c
#include <sys/wait.h> /* For WEXITSTATUS */

void main(){
    int outcome = system("echo ciao"); // execute command in shell
    printf("Outcome = %d\n",outcome);
    outcome = system("if [[ $PWD < \"ciao\" ]]; then echo min; fi");
    printf("Outcome = %d\n",outcome);
    outcome = system("notExistingCommand");
    printf("Outcome = %d\n",WEXITSTATUS(outcome));
}
```

Altre system calls: segnali e processi

Ci sono molte altre system calls per la gestione dei processi e della comunicazione tra i processi che saranno discusse più avanti.

Forking

System call “fork”

La syscall principale per il forking è “fork”.

Il forking è la “generazione” di nuovi processi (uno alla volta) partendo da uno esistente.

Un processo attivo invoca la syscall e così il kernel lo “clona” modificando però alcune informazioni e in particolare quelle che riguardano la sua collocazione nella gerarchia complessiva dei processi.

Il processo che effettua la chiamata è definito “padre”, quello generato è definito “figlio”.

fork: elementi clonati e elementi nuovi

Sono clonati gli elementi principali come il PC (Program Counter), i registri, la tabella dei file (file descriptors) e i dati di processo (variabili). Le meta-informazioni come il “pid” e il “ppid” sono aggiornate (al contrario di execve()!).

L’esecuzione procede per entrambi (quando saranno schedulati!) da PC+1 (tipicamente l’istruzione seguente il fork o la valutazione dell’espressione in cui essa è utilizzata):

Prossimo step: `printf`	Prossimo step: assegnamento ad `f`
`fork();` `printf(“\n”);`	`f=fork();` `printf(“\n”);`

Identificativi dei processi

Ad ogni processo è associato un identificativo univoco per istante temporale, sono organizzati gerarchicamente (padre-figlio) e suddivisi in insiemi principali (sessioni) e secondari (gruppi). Anche gli utenti hanno un loro identificativo e ad ogni processo ne sono abbinati due: quello reale e quello effettivo (di esecuzione)

- PID - Process ID
- PPID - Parent Process ID
- SID - Session ID
- PGID - Process Group ID
- UID/RUID - (Real) User ID
- EUID - Effective User ID

Approfonditi in un'altra lezione!

getpid(), getppid()

`pid_t getpid()` : restituisce il PID del processo attivo
`pid_t getppid()` : restituisce il PID del processo padre

```c
#include <stdio.h> <unistd.h> <stdlib.h>                    //ppid.c

void main(){
    printf("Subshell $$ = ");
    fflush(stdout); // Forza l'output di printf
    system("echo $$"); // subshell
    printf("PID: %dPPID: %d\n",getpid(),getppid());
}
```

(includendo `<sys/types.h>` e `<sys/wait.h>`: pid_t è un intero che rappresenta un id di processo)

fork: valore di ritorno

La funzione restituisce un valore che solitamente è catturato in una variabile (o usato comunque in un'espressione).

Come per tutte le syscall in generale, il valore è -1 in caso di errore (in questo caso non ci sarà nessun nuovo processo, ma solo quello che ha invocato la chiamata).

Se ha successo entrambi i processi ricevono un valore di ritorno, ma questo è diverso nei due casi:

- Il processo padre riceve come valore il nuovo PID del processo figlio
- Il processo figlio riceve come valore 0

fork: relazione tra i processi

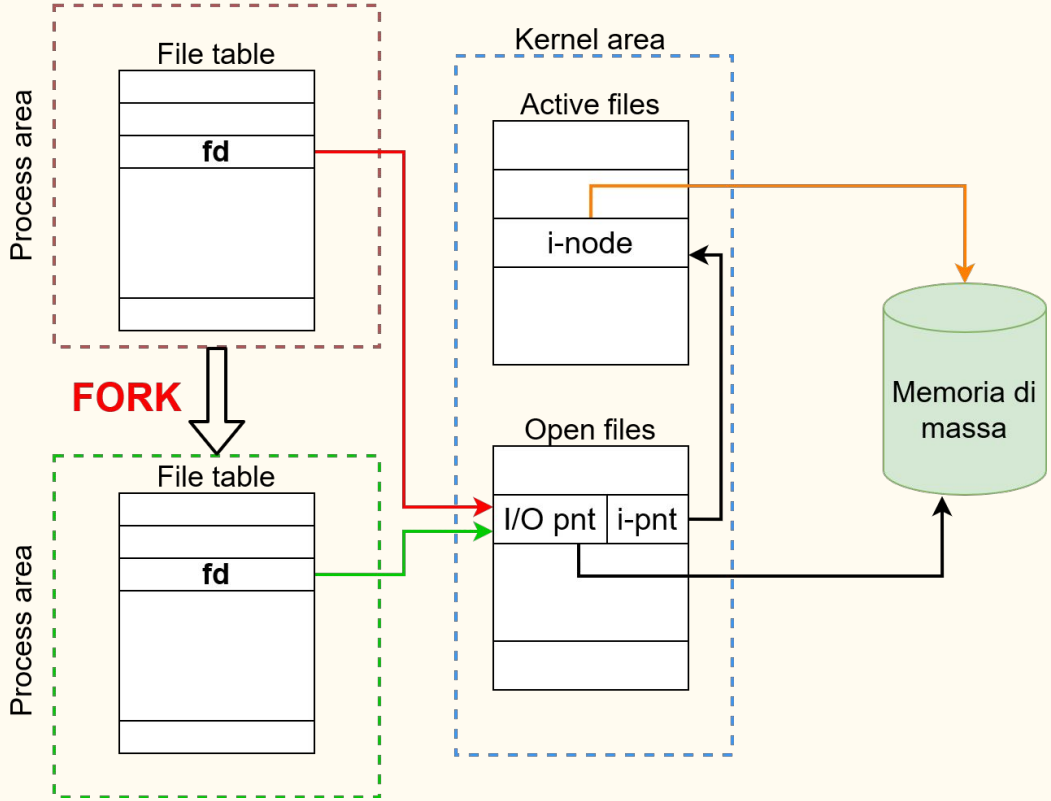
I processi padre-figlio:

- Conoscono reciprocamente il loro PID (ciascuno conosce il proprio tramite `getpid()`, il figlio conosce quello del padre con `getppid()`, il padre conosce quello del figlio come valore di ritorno di `fork()`)

- Si possono usare altre syscall per semplici interazioni come `wait` e `waitpid`

- Eventuali variabili definite prima del fork sono valorizzate allo stesso modo in entrambi: se riferiscono risorse (ad esempio un “file descriptor” per un file su disco) fanno riferimento esattamente alla stessa risorsa.

File Descriptors con fork



fork: wait(), waitpid()

`pid_t wait (int *status)` : attende la **terminazione** di UN figlio (qualunque) e ne restituisce il PID, riportando lo status nel puntatore passato come argomento (se non NULL).

`pid_t waitpid(pid_t pid, int *status, int options)` : analoga a wait ma consente di passare delle opzioni e si può specificare come pid:

- `-n` (<-1: attende un qualunque figlio il cui "gruppo" è |-n|)
- `-1` (attende un figlio qualunque)
- `0` (attende un figlio con lo stesso "gruppo" del padre)
- `n` (>0: attende il figlio il cui pid è esattamente n)

NOTE:
`wait(st)` corrisponde a `waitpid(-1, st, 0)`
`while(wait(NULL)>0);` # attende tutti i figli

Wait: interpretazione stato

Lo stato di ritorno è un numero che comprende più valori “composti” interpretabili con apposite macro, molte utilizzabili a mo’ di funzione (altre come valore) passando lo “stato” ricevuto come risposta come ad esempio:

`WEXITSTATUS(sts)`: restituisce lo stato vero e proprio (ad esempio il valore usato nella “exit”).
`WIFCONTINUED(sts)`: true se il figlio ha ricevuto un segnale SIGCONT.
`WIFEXITED(sts)`: true se il figlio è terminato normalmente.
`WIFSIGNALED(sts)`: true se il figlio è terminato a causa di un segnale non gestito.
`WIFSTOPPED(sts)`: true se il figlio è attualmente in stato di “stop”.
`WSTOPSIG(sts)`: numero del segnale che ha causato lo “stop” del figlio.
`WTERMSIG(sts)`: numero del segnale che ha causato la terminazione del figlio.

Esempio fork multiplo

Ovviamente è possibile siano presenti più “fork” dentro un codice.

Quante righe saranno generate in output dal seguente programma?

```
#include <stdio.h>                    //fork1.c
#include <unistd.h>
int main() {
  fork(); fork(); fork();
  printf("hello\n");
  return 0;
}
```

Esempio fork&wait

```c
#include <stdio.h> <stdlib.h> <unistd.h> <time.h> <sys/wait.h>          //fork2.c
int main() {
  int fid=fork(), wid, st, r; // Generate child
  srand(time(NULL)); // Initialise random
  r=rand()%256; // Get random between 0 and 255
  if (fid==0) {  //If it is child
     printf("Child... (%d)", r); fflush(stdout);
     sleep(3);  // Pause execution for 3 seconds
     printf(" done!\n");
     exit(r); // Terminate with random signal
  } else { // If it is parent
     printf("Parent...\n");
     wid=wait(&st);  // wait for ONE child to terminate
     printf("...child's id: %d==%d (st=%d)\n", fid, wid, WEXITSTATUS(st));
  }
}
```

I processi “zombie” e “orfani”

Normalmente quando un processo termina il suo stato di uscita è “catturato” dal padre: alla terminazione il sistema tiene traccia di questo insieme di informazioni (lo stato) fino a che il padre le utilizza consumandole (con *wait* o *waitpid*). Se il padre non cattura lo stato d’uscita i processi figli sono definiti “zombie” (in realtà non ci sono più, ma esiste un riferimento in sospeso nel sistema).

Se un padre termina prima del figlio, quest’ultimo viene definito “orfano” e viene “adottato” dal processo principale (tipicamente “init” con pid pari a 1).

Un processo zombie che diventa anche orfano è poi gestito dal processo che lo adotta (che effettua periodicamente dei *wait*/*waitpid* appositamente)

Esercizi

Scrivere dei programmi in C che:

1. Avendo come argomenti dei “binari”, si eseguono con *exec* ciascuno in un sottoprocesso (*)
2. idem punto 1 ma in più salvando i flussi di *stdout* e *stderr* in un unico file (*)
3. Dati due eseguibili come argomenti del tipo *ls* e *wc* si eseguono in due processi distinti: il primo deve generare uno *stdout* redirezionato su un file temporaneo, mentre il secondo deve essere lanciato solo quando il primo ha finito leggendo lo stesso file come *stdin*.
   Ad esempio `./main ls wc` deve avere lo stesso effetto di `ls | wc`.

   Suggerimenti:
   - anziché due figli usare padre-figlio
   - usare dup2 per far puntare il file-descriptor del file temporaneo su *stdout* in un processo e *stdin* nell’altro
   - sfruttare *wait* per attendere la conclusione del processo che genera l’output

(*) generando DUE figli

CONCLUSIONI

Tramite l'uso dei *file-descriptors*, di *fork* e della famiglia di istruzioni *exec* è possibile generare più sottoprocessi e "redirezionare" i loro canali di in/out/err.

Sfruttando anche *wait* e *waitpid* è possibile costruire un albero di processi che interagiscono tra loro (non avendo ancora a disposizione strumenti dedicati è possibile sfruttare il file-system - ad esempio con file temporanei - per condividere informazioni/dati).