LabSO 2024

Laboratorio Sistemi Operativi - A.A. 2023-2024

Michele Grisafi - michele.grisafi@unitn.it

Examples of system calls

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>

int 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>

int main(){
    char s[100];
    getcwd(s,100); // copy path in buffer
    printf("%s\n", s); //Print current working dir
    chdir(".."); //Change working dir of the calling process
    char * newBuffer = getcwd(NULL,0); // Allocates buffer if buf is NULL
    printf("%s\n", newBuffer);
    free(newBuffer);
}
```

Operazioni con i file

```
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
...
int remove(const char *pathname);
```

Rimuove un file o una directory (chiamando unlink() o rmdir())

```
int creat(const char *path, mode_t mode);
```

Usa la `open()` per creare/sovrascrivere un file, il cui file descriptor viene restituito in WRONLY

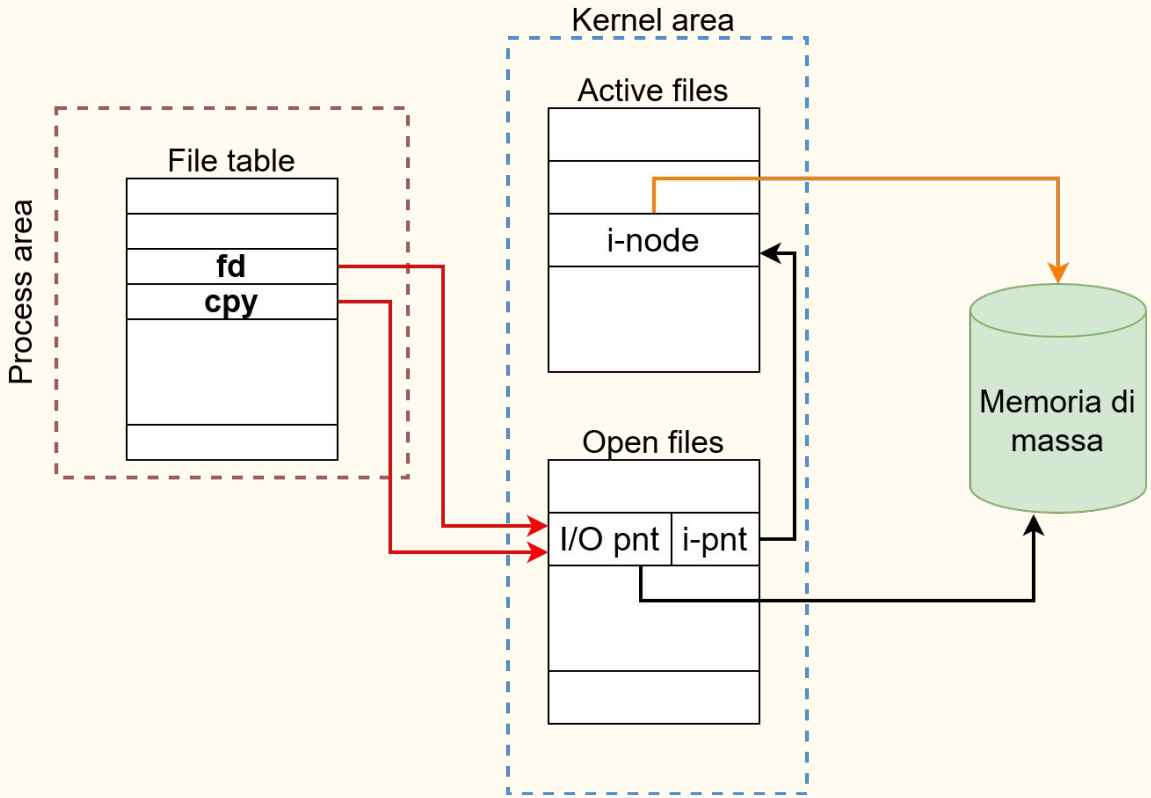
Duplicazione file descriptors: dup(), dup2()

```
int dup(int oldfd);
int dup2(int oldfd, int newfd); //close newfd if open
```

Duplicando un file descriptor avremo un altro riferimento per accedere al medesimo file. Il nuovo file descriptor condivide locks, I/O pointer, flags (tranne close-on-exec).

Si possono usare per modificare stdin, stdout e stderr.

Duplicazione file descriptors: dup(), dup2()



Esempio

```
#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 at most 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",15); // Write starting from 0-pos
    close(cpy); //Close only ONE file descriptor
    write(cpy,"New content using cpy\n",strlen(“New …”)); //Error!
    write(cpy,"New content using 22\n",strlen(“New …”)); //Working!
}
```

Esempio

```
#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 at most 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,1); // Copy cpy to descriptor 1 (closing stdout)
    write(cpy,"Hello",5); //Write to file
    printf(" world\n"); //Write to file!
}
```

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)
```

```
#include <fcntl.h> <unistd.h> <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
}
```

Se non si esegue con sudo (o da root), si potrà cambiare solo il gruppo in uno di quelli a cui l'utente che esegue il comando appartiene!

Eseguire programmi: la famiglia **exec**

La famiglia di funzioni exec ha come scopo l'esecuzione di un programma, sostituendo l'immagine del processo corrente con una nuova immagine. Il PID del processo e la sua file table non cambiano*! La system call di base è `execve()`, ma esistono vari alias che differiscono solo per gli argomenti accettati. Ogni alias è composto dalla parola chiave `exec` seguita dalle seguenti lettere:

- '`l`': accetta una lista di argomenti
- '`v`': accetta un vettore, quindi un solo argomento di diversi argomenti
- '`p`': usa la variabile d'ambiente `PATH` per cercare il binario
- '`e`': usa un vettore di variabili d'ambiente (es. "`name=value`")

NB: Ogni vettore di argomenti deve terminare con un elemento NULL!

```
int execv    (const char *path, char *const argv[])
int execvp   (const char *file, char *const argv[])
int execve   (const char *path, char *const argv[],
              char *const envp[]);
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 *path, const char * arg0,…,argn,NULL,
            char *const envp[])
int execlpe(const char *file, const char * arg0,…,argn,NULL,
            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()

```
#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");
}
```

```
#include <stdio.h>                                           //execle2.out
#include <stdlib.h>
void main(int argc, char ** argv){
    printf("This is execle2 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#3
  execvp(argv[0],argv); // Replace current process
}
```

Chiamare la shell: system()

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

Esegue il comando specificato in una shell di sistema (sh) e si comporta come un sottoprocesso che invoca `execl("/bin/sh", "sh", "-c", command, (char *) NULL);`
Il valore di ritorno è il seguente:

- Se ‘command’ è NULL, restituisce non-zero se una shell è disponibile, 0 altrimenti
- Se un processo figlio non è stato creato, -1
- Se una shell non può essere invocata allora restituisce un valore equivalente all’aver chiamato exit(127) sulla shell
- Se la chiamata riesce, restituisce lo stato della shell chiamata.

Negli ultimi due casi, il valore di ritorno va interpretato con le direttive WIFEXITED(), WEXITSTATUS(), (che vedremo nelle prossime slides).

Esempio: system()

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

void main(){
    int outcome = system("echo ciao ; echo $0");
    printf("Outcome = %d %d\n",outcome,WEXITSTATUS(outcome));
    outcome = system("notExistingCommand");
    printf("Outcome = %d\n",WEXITSTATUS(outcome));
    outcome = system("exit 23");
    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”

Il forking è la “generazione” di nuovi processi (uno alla volta) partendo da uno esistente. La syscall principale per il forking è “`fork()`”.

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

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

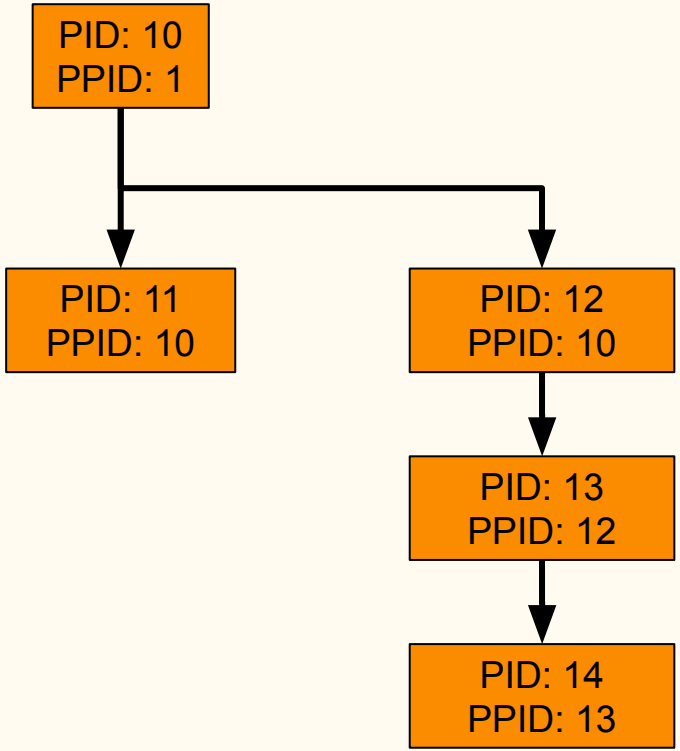
Identificativi dei processi

Ad ogni processo è associato un identificativo univoco per istante temporale. I processi sono organizzati gerarchicamente (genitore-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!

Gerarchia: genitore-figlio



getpid(), getppid()

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

```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)

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”);`

fork: valore di ritorno

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

Come per molte syscall, 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 genitore riceve come valore il nuovo PID del processo figlio
- Il processo figlio riceve come valore 0

Esempio

```
#include <stdio.h>   <unistd.h>
int main(void) {
    int result = fork();
    if(result == 0){
        printf("I’m the child\n");
        return 0;
    }
    else{
        printf("I’m the parent\n");
    }
    return 0;
}
```

```
int isChild = !fork();
int isParent = fork()!=0
```

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;
}
```

fork: relazione tra i processi

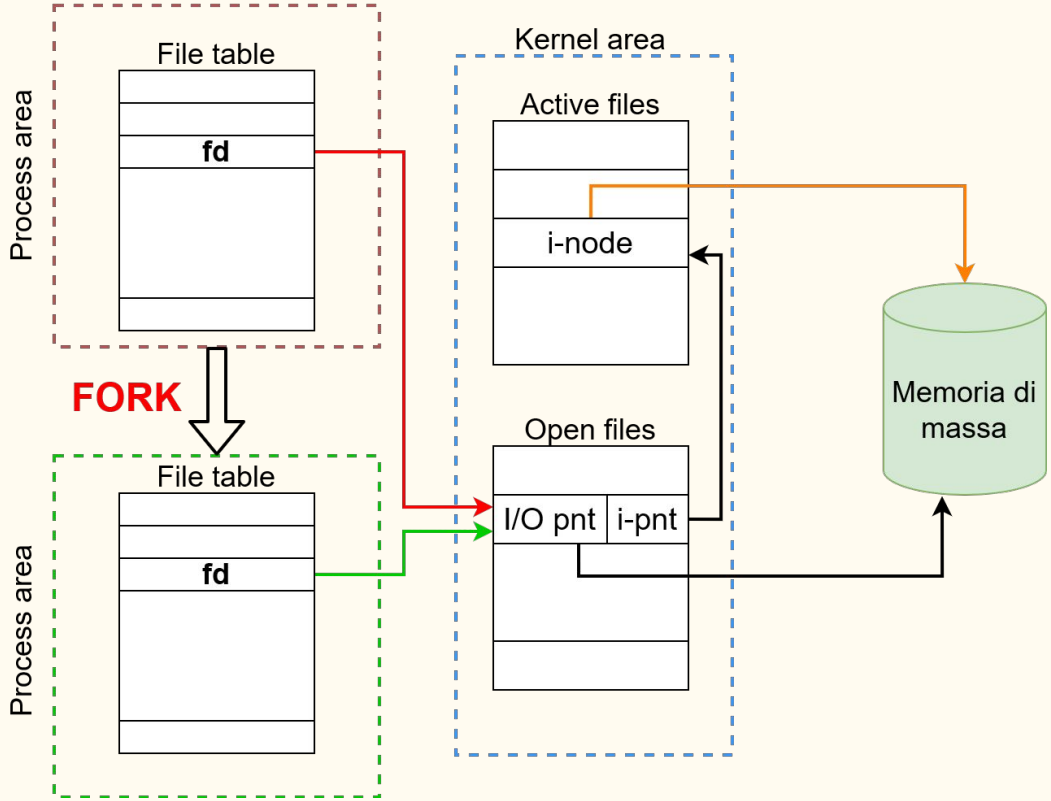
I processi genitore-figlio:

- Conoscono reciprocamente il loro PID (ciascuno conosce il proprio tramite `getpid()`, il figlio conosce quello del genitore con `getppid()`, il genitore 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 e diventano poi indipendenti! Non c'è condivisione alcuna tra i due processi!
- Risorse, ad esempio un "file descriptor" per un file su disco, fanno riferimento esattamente alla stessa risorsa.

Esempio variabili

```
int main(void){
    int counter = 0;
    if(fork()==0){
        counter++;
        printf("Counter in child = %d\n",counter);
    }else{
        counter = 10;
        printf("Counter in parent = %d\n",counter);
    }
    printf("Counter for both = %d\n",counter); //Both will print their value
}
```

File Descriptors con fork



Esempio file

```
int main(void){
    int fd = open("/tmp/fd.txt",O_CREAT|O_WRONLY|O_TRUNC,S_IRUSR | S_IWUSR);
    write(fd,"scritto unico\n",strlen("scritto unico\n"));
    if(fork()==0){ //Child
        write(fd,"scritto dal figlio\n",strlen("scritto dal figlio\n"));
        lseek(fd,0,SEEK_SET);
    }else{ //Parent
        sleep(1);
        write(fd,"scritto dal padre\n",strlen("scritto dal padre\n"));
    }
    return 0;
}
```

fork: wait()

`pid_t wait (int *status)`

Attende il **cambio di stato** di un processo figlio (uno qualsiasi) restituendone il PID e salvando in `status` lo stato del figlio (se il puntatore non è `NULL`). Il cambio di stato avviene se il figlio viene terminato o la sua esecuzione interrotta/ripresa a seguito di un segnale (prossime lezioni) Se non esiste alcun figlio restituisce -1.

Nel nostro caso ci interessa principalmente la terminazione del figlio. Questa system call ci permette di bloccare il processo (anche sincronizzarlo) fino a quando il figlio non ha finito le sue operazioni.

`while(wait(NULL)>0);` questo comando aspetta 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

```
#include <stdio.h> <unistd.h> <sys/wait.h>
int main(void) {
    int isChild = !fork();
    if(isChild){
        sleep(3); return 5;
    }
    int childStatus;
    wait(&childStatus);
    printf("Children terminated? %d\nReturn code: %d\n",
        WIFEXITED(childStatus),WEXITSTATUS(childStatus));
    return 0;
}
```

fork: waitpid()

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

Consente un'attesa selettiva basata su dei parametri. `pid` può essere:

- `-n` (aspetta un figlio qualsiasi nel gruppo `|-n|`) (prossime lezioni)
- `-1` (aspetta un figlio qualsiasi)
- `0` (aspetta un figlio qualsiasi appartenente allo stesso gruppo)
- `n` (aspetta il figlio con `PID=n`)

`options` sono i seguenti parametri ORed:

- `WNOHANG`: ritorna immediatamente se nessun figlio è terminato → non si resta in attesa!
- `WUNTRACED`: ritorna anche se un figlio si è interrotto senza terminare.
- `WCONTINUED`: ritorna anche se un figlio ha ripreso l'esecuzione.

`wait(&st)` è l'equivalente di `waitpid(-1, &st, 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 viene “catturato” dal genitore: alla terminazione il sistema tiene traccia di questo insieme di informazioni (lo stato) fino a che il genitore le utilizza consumandole (con `wait` o `waitpid`). Se il genitore non cattura lo stato d’uscita, i suoi processi figli vengono definiti “zombie” (in realtà non ci sono più, ma esiste un riferimento in sospeso nel sistema).

Se un genitore termina prima del figlio, quest’ultimo viene definito “orfano” e viene “adottato” dal processo principale (tipicamente “systemd” 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).

I processi “zombie” e “orfani”

Per ispezionare la lista di processi attivi usare il comando ‘`ps`’ con le seguenti opzioni:

- a: mostra lo stato (T: stopped, Z: zombie, R: running, etc...)
- -H: mostra la gerarchia processi
- -e: mostra l’intera lista dei processi, non solo della sessione corrente
- -f: mostra il PID del padre

```
ps a -Hef
```

Esempio

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

int main(void){
     if(fork()==0){
          return 1; //Terminate the child
     }else{
          if(fork()==0) while(1);
          sleep(10); //Sleep → execute “ps a -Hf”
          wait(NULL); // execute “ps a -Hf” again
          sleep(10);
     }
     return 0; // execute “ps a -Hf” again
}
```

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`.

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).