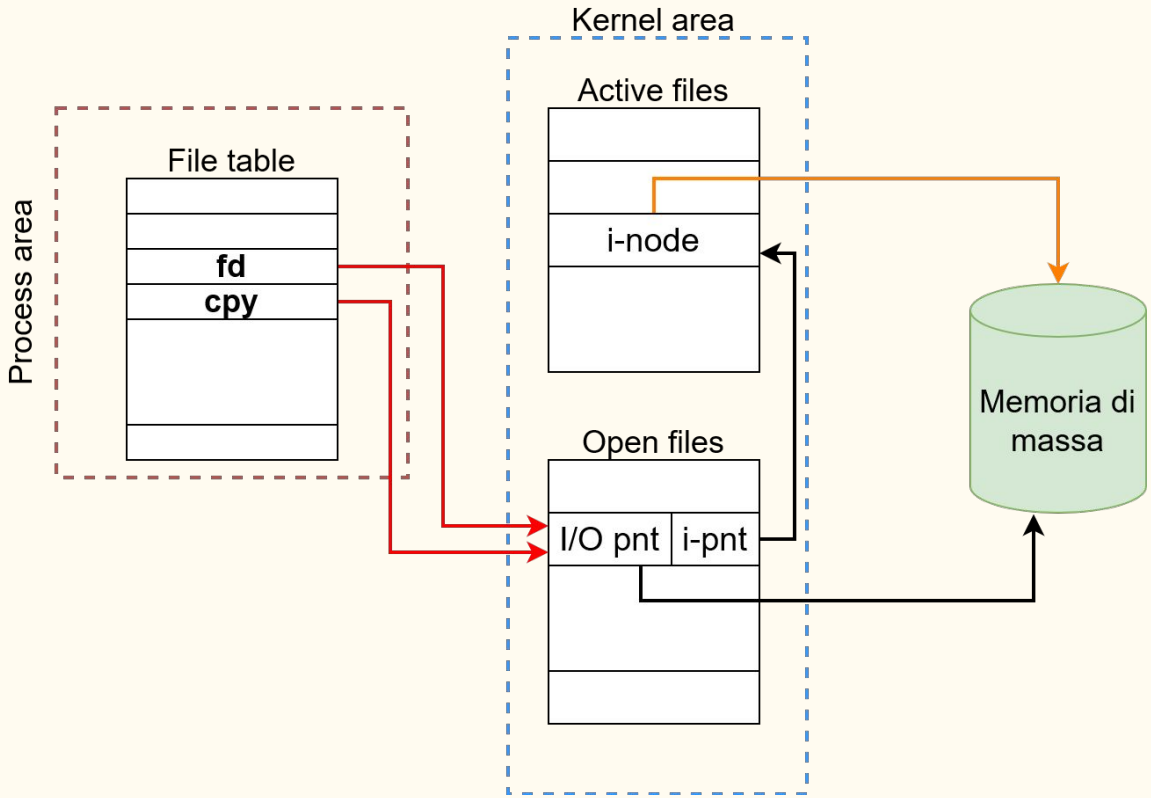
Duplicazione file descriptors: dup(), dup2()



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)
```

```
#include <fcntl.h>              //execute with sudo!    //chown.c
#include <unistd.h>
#include <sys/stat.h>
void main(){
    int fd = open("file",O_RDONLY);
    fchown(fd, 0, 0); // Change owner to root:root
    chmod("file",S_IRUSR|S_IRGRP|S_IROTH); // Permission to r/r/r
}
```

Eseguire programmi: exec_()

```
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()

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

void main(){
    // '/bin/sh -c string'
    int outcome = system("echo ciao"); // execute command in shell
    printf("%d\n",outcome);
    outcome = system("if [[ $PWD < \"ciao\" ]]; then echo min; fi");
    printf("%d\n",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.

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

fork: getpid(), getppid()

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

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

void main(){
    printf("$$ = ");
    fflush(stdout);
    system("echo $$"); // subshell
    printf("  PID: %d  PPID: %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

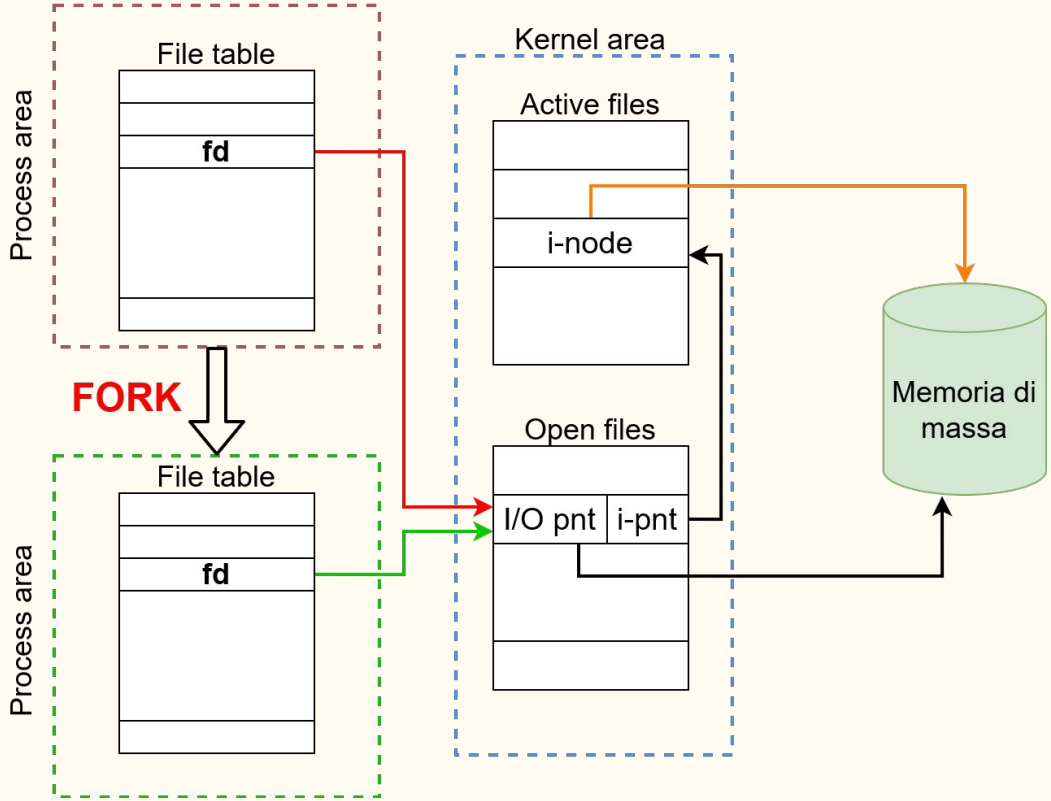
I processi padre-figlio:

- 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 conclusione di UN figlio 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` (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
  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 a tale scopo fino a che il padre le utilizza (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 nel sistema in sospeso).

Se un padre termina prima del figlio, quest’ultimo è 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).