Sistemi Operativi

Laboratorio – linea 2

3

Matteo Re

Dip. di Informatica

Università degli studi di Milano

matteo.re@unimi.it

http://homes.di.unimi.it/re/solabL2.html

Sistemi Operativi

Laboratorio – linea 2

3

Lezione 3:

Shell
Processi
Thread
Pipe
File

Nella lezioni precedenti abbiamo cercato di rispondere alla domanda :

“Perchè serve un sistema operativo?”

Un s.o. rende molto più conveniente l'utilizzo della macchina fisica.

Per riuscirci il s.o. :

- fornisce un insieme di astrazioni che **semplificano l'uso di periferiche e memoria**.
- Gestisce opportunamente le risorse fra tutte le attività in corso.

Abbiamo inoltre visto come il s.o. **fornisce** alcune di queste astrazioni (dallo hardware) mediante le **system call** .

Sistemi Operativi

Laboratorio – linea 2

3

Filo conduttore del corso sono le astrazioni fornite dal sistema operativo. Le principali sono:

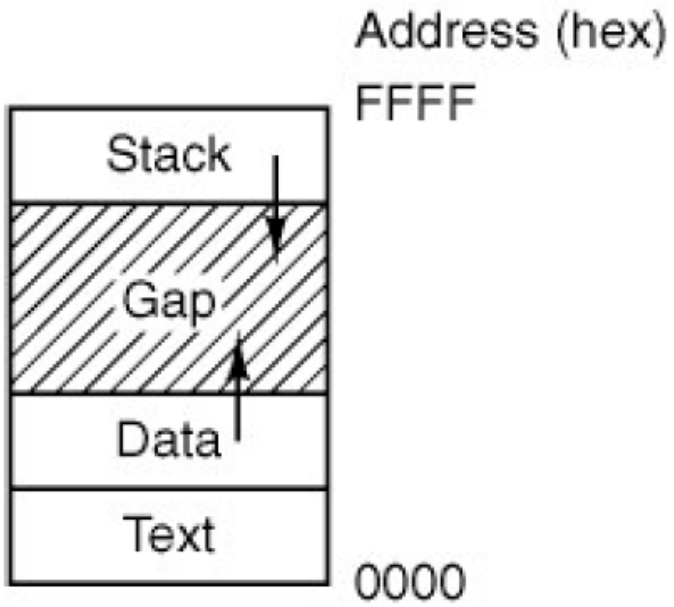
- System call (lez. 2)
- Memoria virtuale
- Processo
- File
- Shell

MEMORIA VIRTUALE

Ogni programma ha a disposizione un **blocco di indirizzi di memoria** per la sua esecuzione. Nei sistemi a 32 bit la dimensione di questo blocco di memoria (virtuale) è **4 Gb**.

MEMORIA VIRTUALE

Generalmente la memoria virtuale è divisa in segmenti: Text (codice, read only), dati inizializzati, stack e heap.



PROCESSI E THREAD

- **Programma** : un programma è la codifica di un ***algoritmo*** in una forma eseguibile da una macchina.
- **Processo :** è un programma **in esecuzione**
- **Thread** (filo conduttore) è una **sequenza di istruzioni in esecuzione**: più thread possono condividere lo spazio di memoria in cui le istruzioni lavorano. Il termine assume anche una accezione tecnica nei sistemi operativi che distinguono le due astrazioni.

Ogni processo dà vita ad **almeno** un thread. Ogni CPU può eseguire, in un dato momento **al più** un thread.

POSIX syscall (process management)

pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = wait(&status)	Old version of waitpid
s = execve(name, argv, envp)	Replace a process core image
exit(status)	Terminate process execution and return status
size = brk(addr)	Set the size of the data segment
pid = getpid()	Return the caller's process id
pid = getpgrp()	Return the id of the caller's process group
pid = setsid()	Create a new session and return its process group id
l = ptrace(req, pid, addr, data)	Used for debugging

CREAZIONE DI PROCESSI mediante fork()

Il processo creante è detto **parent** il processo creato è detto **child** . Dato che ogni processo (a parte init) ha un processo parent si determina una struttura di “parentela” ad albero.

Nelle prossime slide vedremo come utilizzare la shell per osservare le relazioni di parentela tra I processi.

Caratteri utili:

| (pipe) alt+124
& alt+38
; alt+59

SHELL

La shell è l'*interprete dei comandi* che l'utente dà al sistema operativo. Ne esistono grafiche e testuali.
In ambito GNU/Linux la piú diffusa è una shell testuale `bash`, che fornisce i costrutti base di un linguaggio di programmazione (variabili, strutture di controllo) e primitive per la gestione dei processi e dei file.

CREAZIONE DI PROCESSI

La creazione di processi richiede alcune operazioni da parte del s.o. :

- Creazione nuovo ID (PID – Process Identifier)
- Allocazione Memoria (Codice, Dati)
- Allocazione altre risorse (stdin, stdout, dispositivi I/O in genere)
- Gestione informazioni sul nuovo processo (es. priorità)
- Creazione PCB (Process Control Block) contenente le informazioni precedenti

In s.o. della famiglia Unix ogni processo viene creato da un altro processo. Unica eccezione: **init** (PID=1, è il primo processo … viene creato durante il boot)

Nello standard POSIX la system call per creare processi è **fork** .

SHELL

Il funzionamento di una generica shell obbedisce a questo ciclo:

- Attesa input utente
- Interpretazione comando utente
- Esecuzione comando/i utente

L'implementazione di una shell richiede l'utilizzo della chiamata di sistema **fork**. Ogni comando eseguito dopo aver interpretato l'input dell'utente genera un processo figlio che ha come parent la shell.

SHELL (pseudo codice)

```
1  while (1){ /* repeat forever */
2          type_prompt(); /* display prompt on the screen */
3          read_command(command, parameters); /* read input from terminal */
4          if (fork() > 0){ /* fork off child process */
5                  /* Parent code. */
6                  waitpid(1, &status, 0); /* wait for child to exit */
7          } else {
8                  /* Child code. */
9                  execve(command, parameters, 0); /* execute command */
10         }
11 }
```


Cosa succede **dopo** che un processo è stato creato ?

Relazioni dinamiche :

Dopo la creazione del processo figlio il processo **padre** puo:

1) ***attendere*** la terminazione del processo figlio

linuxprompt$ top

Il terminale è associato al nuovo processo. Per interrompere l'esecuzione del processo figlio: **ctrl+c**

2) ***continuare*** senza attendere (es. esecuzione in background nella shell)

linuxprompt$ sleep 500 **&**

Quando un comando viene eseguito in background al genitore (shell) viene notificato il suo PID

Relazioni dinamiche :

Il comando utilizzato in linux per ottenere informazioni sui processi è **ps**

Provate questi comandi in bash:

ps
ps -o pid,ppid,command
ps --forest -o pid,ppid,command

notate che ps è esso stesso un processo figlio della shell

L'ultimo esempio dovrebbe restituire un output simile al seguente (i PIDs saranno diversi)

```
user@:~$ ps --forest -o pid,ppid,command
  PID  PPID COMMAND
 2153  2111 -bash
 2222  2153  \_ sleep 500
 2227  2153  \_ ps --forest -o pid,ppid,command
```

ps di default mostra solo i processi associati al terminale da cui viene lanciato. In questo caso abbiamo la shell bash, il comando *sleep* e lo stesso ps

Relazioni di contenuto :

Due possibilità:

- Il processo figlio è un duplicato del padre (sistemi Unix)
- Il figlio esegue un programma differente (sistemi Windows)

Questo è il comportamento standard ma entrambe le opzioni sono disponibili sia in sistemi Unix che in sistemi Windows.

System call fork

La chiamata di sistema **fork** permette di creare un processo duplicato del processo genitore.

fork, come la maggior parte delle chiamate di sistema che discuteremo, appartiene allo standard **POSIX** (Portable Operating System Interface) di IEEE (Institute of Electrical and Electronics Engineers). È utilizzabile in qualsiasi sistema che supporti POSIX.

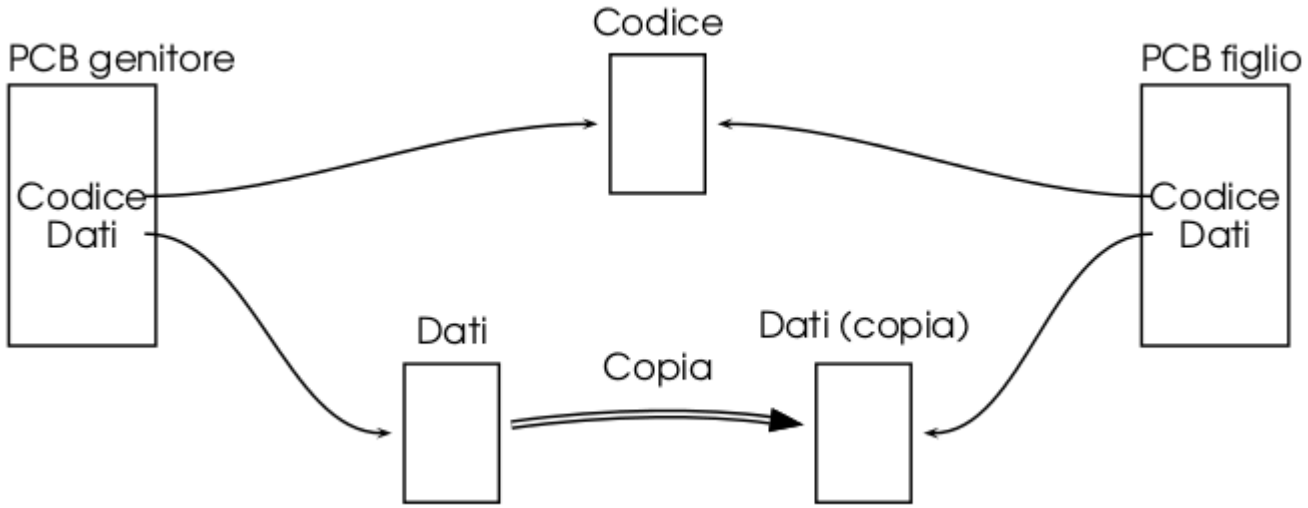
La fork crea un **nuovo** processo che:

- condivide l'area **codice** del processo genitore
- utilizza una copia dell'area **dati** del processo genitore
- ha un PID diverso dal genitore (questo si può sfruttare per fare compiere operazioni differenti al processo padre ed ai processi figli)

Come possiamo differenziare I processi?

Valori di ritorno di fork:

- <0 **Errore**
- =0 Processo figlio
- >0 Processo genitore: il valore di ritorno è il PID del figlio.



Per recuperare sorgenti SOLAB L2 lezione 3:

```
linuxprompt$ wget http://homes.di.unimi.it/re/SYSOPLAB/L3.tar.gz
linuxprompt$ tar -xvzf L3.tar.gz
```

Vale la raccomandazione delle lezioni precedenti. Sforzatevi di scrivere I sorgenti dei programmi che vengono proiettati alla lavagna da soli ma, se non riuscite a farlo in tempo per quando viene compilato e testato il risultato, usate I file forniti nell'archivio tar compresso.

Schema standard di utilizzo di fork (C) :

```
...
if ( (pid = fork() ) < 0 )
      perror("fork error"); // stampa la descrizione dell'errore
else if (pid == 0) {
      // codice figlio
} else {
      // codice genitore, (pid > 0)
}
// codice del genitore e del figlio. Usare con cautela!!!
...
```

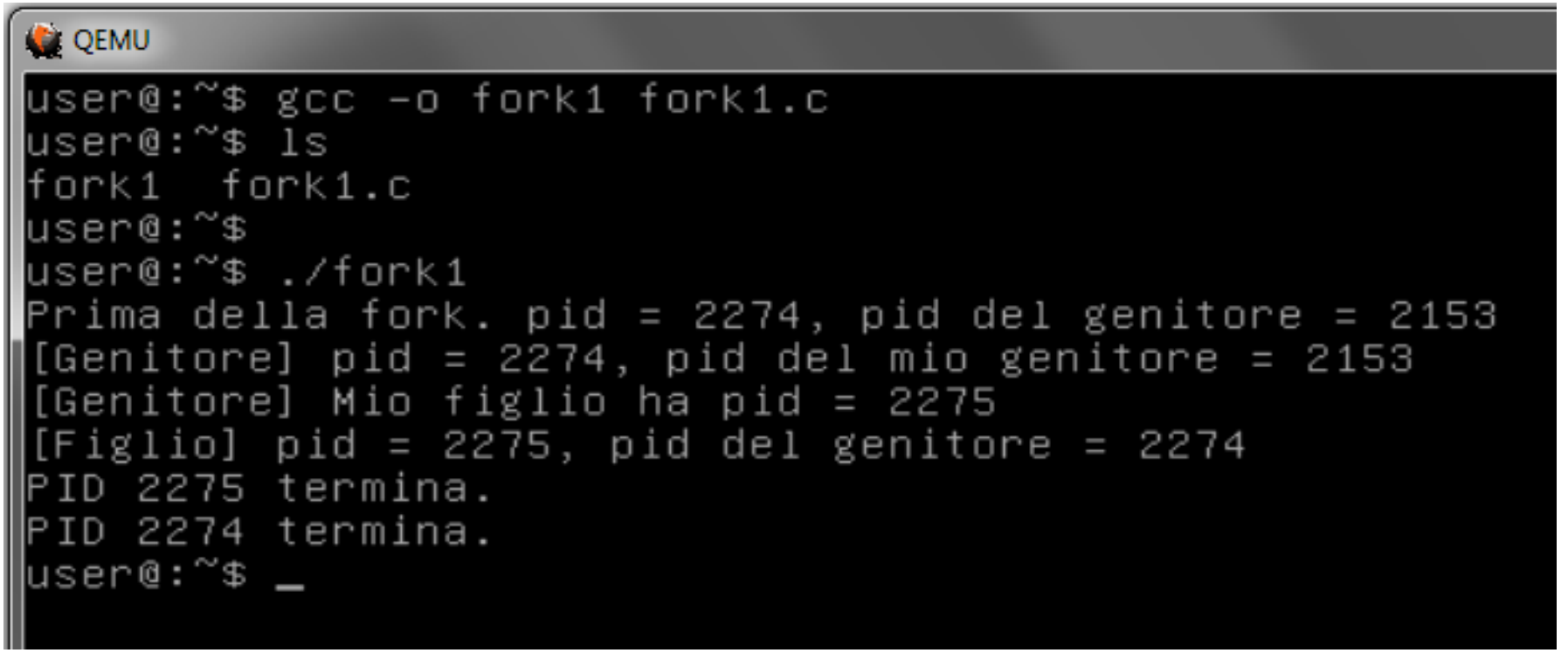
NB: **pid_t** è un *signed integer*, cioè int in molti sistemi ma potrebbe essere di dimensioni differenti, es. long, ed è quindi bene usare sempre il tipo pid_t

fork1.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
main() {
        pid_t pid;

        printf("Prima della fork. pid = %d, pid del genitore = %d\n",getpid(), getppid());

        if ( (pid = fork()) < 0 )
                perror("fork error"); // stampa la descrizione dell'errore
        else if (pid == 0) {          // figlio
                printf("[Figlio] pid = %d, pid del genitore = %d\n",getpid(), getppid());
        } else {                      // genitore
                printf("[Genitore] pid = %d, pid del mio genitore = %d\n",getpid(), getppid());
                printf("[Genitore] Mio figlio ha pid = %d\n",pid);
                  sleep(1); // attende 1 secondo
        }
        // entrambi i processi
        printf("PID %d termina.\n", getpid());
}
```

fork1.c

Compilazione: gcc -o fork1 fork1.c
Run: ./fork1



Nel prossimo esercizio (fork2.c) proveremo a visualizzare alternanza tra processi padre e figlio. Terminate esecuzione con **ctrl+c**

fork2.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdbool.h>
main() {
        pid_t pid;
        int i;

        if ( (pid = fork()) < 0 )
                perror("fork error");           // stampa la descrizione dell'errore
        else if (pid == 0) {
                while(true) {
                        for (i=0;i<100000000;i++) {} // serve per rallentare
                        printf("Figlio: pid = %d, pid del genitore = %d\n",getpid(), getppid());
                }
        } else {
                while(true) {
                        for (i=0;i<100000000;i++) {} // serve per rallentare
                        printf("genitore: pid = %d, pid di mio genitore = %d\n",getpid(), getppid());
                }
        }
}
```

fallimento fork

Una fork fallisce quando non è possibile creare un processo e tipicamente questo accade quando non c'è memoria per il processo o per il kernel.

ATTENZIONE !

Il prossimo test potrebbe **bloccare tutto il sistema** perché i processi generati vanno ad occupare tutte le risorse ed il sistema non può più prendere il controllo. È necessario **limitare il numero di processi utenti** tramite il comando **ulimit -u 300** (al massimo 300 processi sono ammessi per l'utente sul presente terminale). Attenzione che il limite **è per terminale** quindi il test va eseguito dalla stessa finestra su cui avete impostato il limite a 300.

ffailure.c

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

main() {
        while(1){
            if(fork() < 0){
                perror("errore fork");
            }
        }
}
```

Compilazione: gcc -o ffailure ffailure.c
Esecuzione: linuxprompt$ **ulimit -u 300**
linuxprompt$ ./ffailure

ESERCIZIO 3.1 :
Quanti processi eseguono la fork al loop i-esimo? (pensate a quanti processi ci sono alla prima fork, quanti alla seconda, e cosi' via). Dopo aver risposto provate a modificare il sorgente per dimostrare l'esattezza della risposta.

Processi orfani

Se inseriamo una sleep subito prima della printf nel processo figlio lo rendiamo **orfano** perché il processo genitore termina prima di lui:

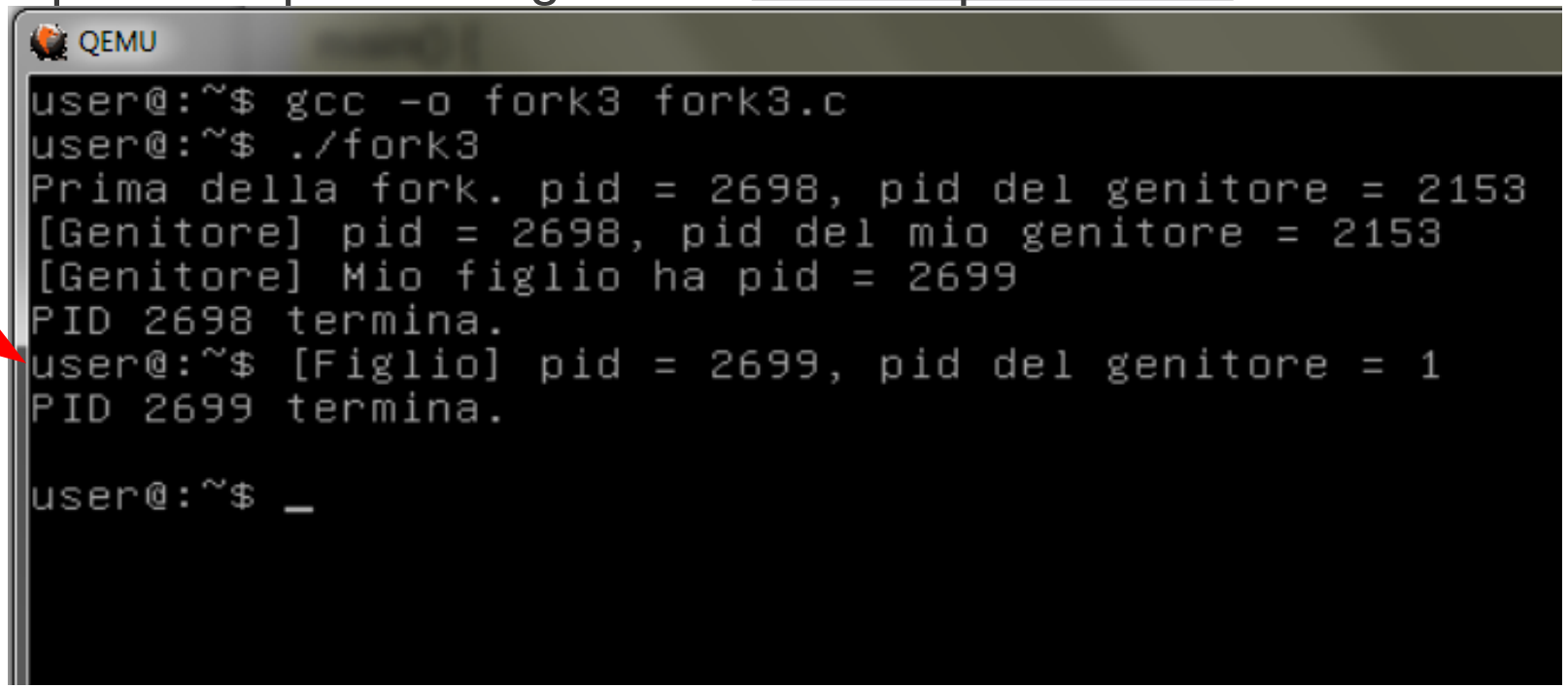
fork3.c
(modifica di fork1.c)

```
main() {
        pid_t pid;

        printf("Prima della fork. pid = %d, pid del genitore = %d\n",getpid(), getppid());

        if ( (pid = fork()) < 0 )
                perror("fork error"); // stampa la descrizione dell'errore
        else if (pid == 0) {          // figlio
                sleep(5);
                printf("[Figlio] pid = %d, pid del genitore = %d\n",getpid(), getppid());
        } else {                      // genitore
                printf("[Genitore] pid = %d, pid del mio genitore = %d\n",getpid(), getppid());
                printf("[Genitore] Mio figlio ha pid = %d\n",pid);
                sleep(1); // attende 1 secondo
        }
        // entrambi i processi
        printf("PID %d termina.\n", getpid());
}
```

Processi orfani

Se inseriamo una sleep subito prima della printf nel processo figlio lo rendiamo **orfano** perché il processo genitore termina prima di lui:

NB: Il prompt viene restituito quando termina il processo genitore..



DOMANDA: A quanto pare il processo figlio è stato “adottato”. **Da chi**?
NOTA: Un processo orfano non viene più terminato da ctrl+c o dalla chiusura della shell (provare ctrl+c).

Processi zombie

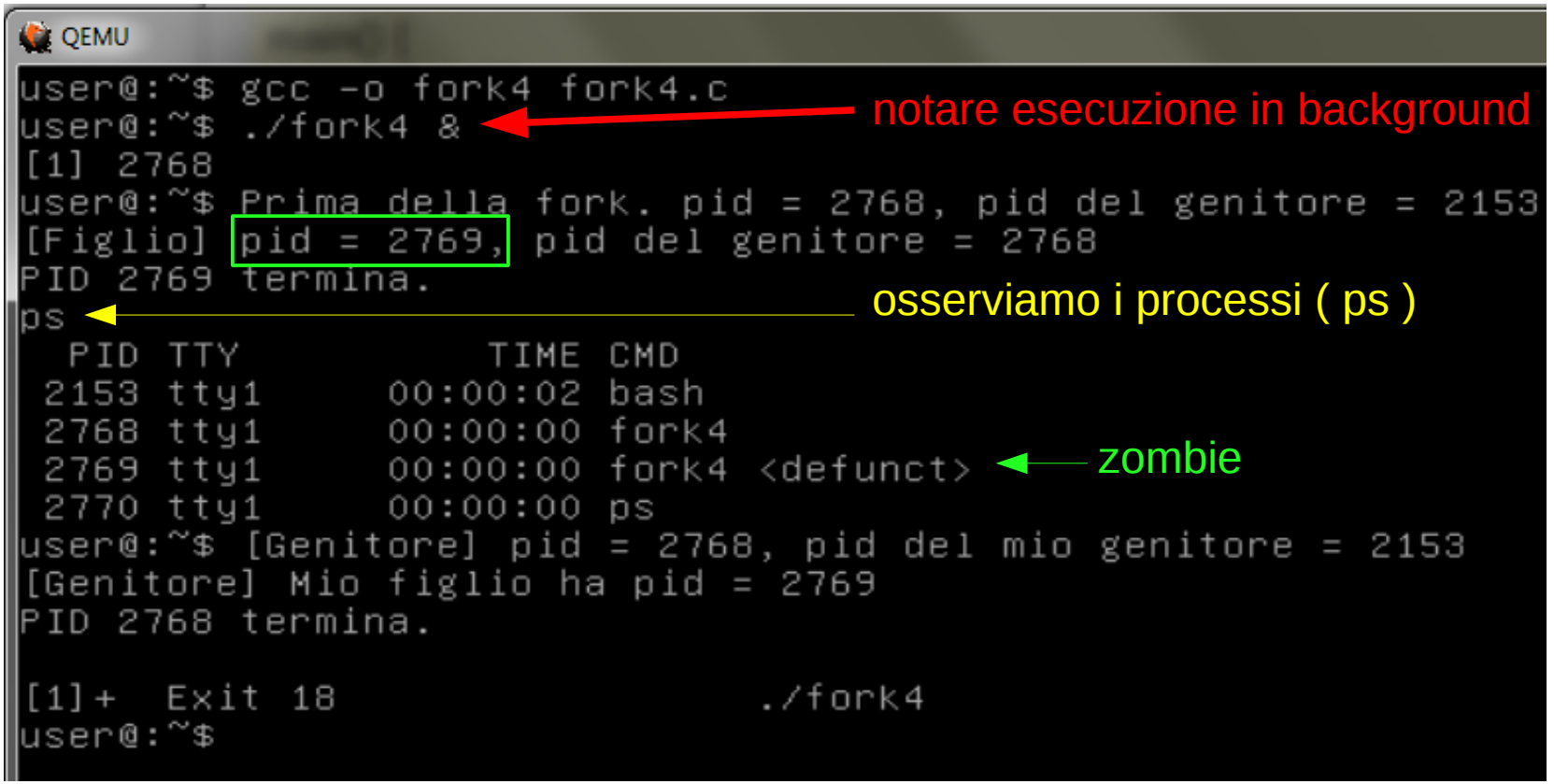
Gli *zombie* sono processi terminati ma in attesa che il genitore rilevi il loro stato di terminazione. Per osservare la generazione di un processo zombie ci basta porre la sleep prima della printf del processo genitore:

fork4.c
(modifica di fork1.c)

```
main() {
        pid_t pid;

        printf("Prima della fork. pid = %d, pid del genitore = %d\n",getpid(), getppid());

        if ( (pid = fork()) < 0 )
                perror("fork error"); // stampa la descrizione dell'errore
        else if (pid == 0) {          // figlio
                printf("[Figlio] pid = %d, pid del genitore = %d\n",getpid(), getppid());
        } else {                      // genitore
                sleep(5);
                printf("[Genitore] pid = %d, pid del mio genitore = %d\n",getpid(), getppid());
                printf("[Genitore] Mio figlio ha pid = %d\n",pid);
                sleep(1); // attende 1 secondo
        }
        // entrambi i processi
        printf("PID %d termina.\n", getpid());
}
```

Processi zombie

Per effettuare questo test dovrete essere **molto veloci**! Non appena vedete il primo messaggio di terminazione processo (quello del figlio) scrivere ps e premete invio.



Esecuzione e termimazione

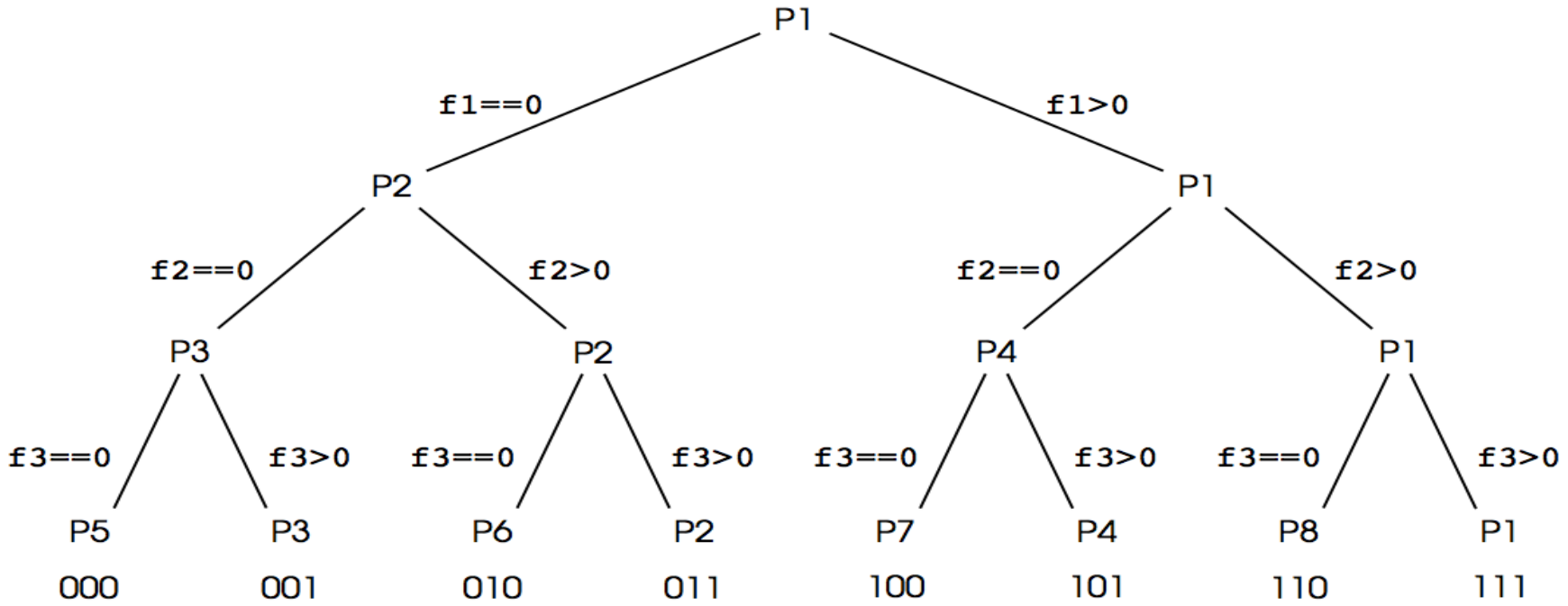
Lo scopo del seguente esercizio è la comprensione approfondita del funzionamento della fork e, in particolare, del fatto che dopo ogni fork esiste un processo identico al processo genitore (tranne che per il **valore di ritorno** della fork) in esecuzione nello stesso punto del programma.

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

main() {
  pid_t f1,f2,f3;

  f1=fork();
  f2=fork();
  f3=fork();

  printf("%i%i%i ", (f1 > 0),(f2 > 0),(f3 > 0));
}
```


Domanda: secondo voi che output produce? Perchè?

Permutazione dei seguenti valori : **000 001 100 101 010 011 110 111**

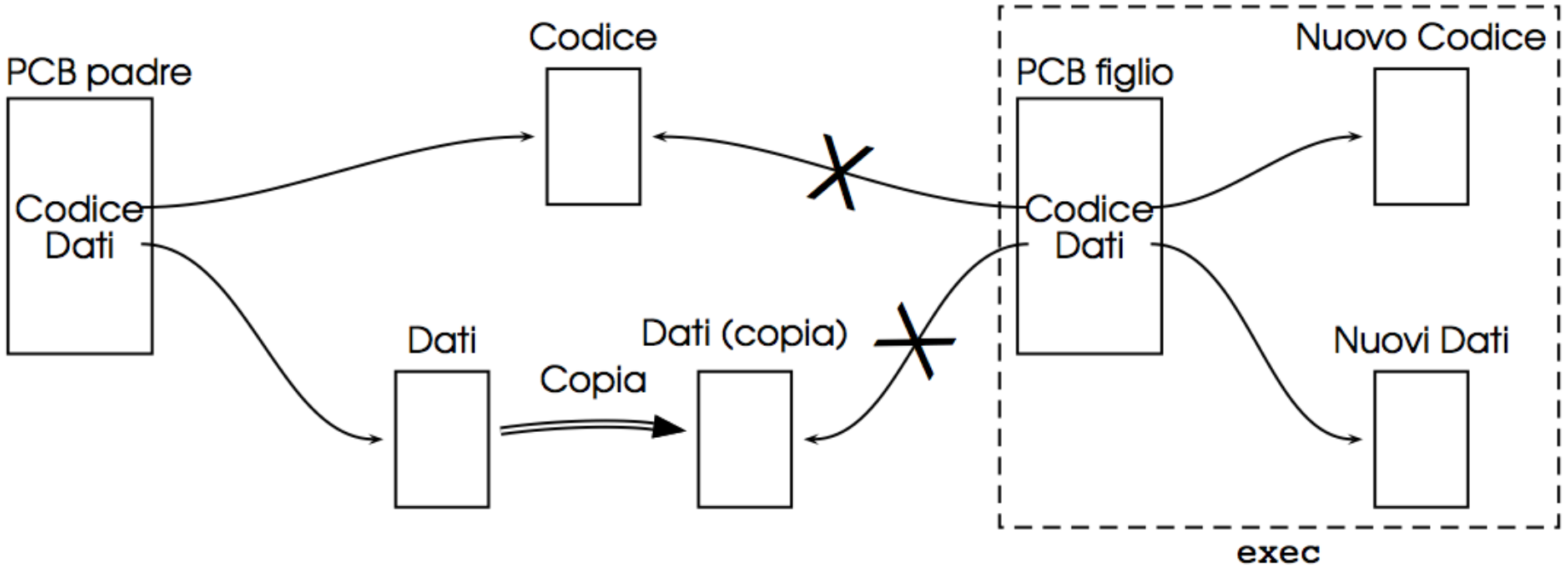
ossia tutti i numeri binari di 3 cifre in qualche ordine (dipende dallo scheduling). Perchè succede? Possiamo visualizzarlo con un albero binario in cui mettiamo a destra il processo genitore (stesso id del nodo genitore) e a sinistra il processo figlio generato dalla fork.
Il valore di f1, f2 ed f3 sono quindi 0 sul ramo di sinistra e >0 sul ramo di destra.



System call exec

La chiamata di sistema exec permette di eseguire un programma diverso da quello che ha effettuato la fork. exec sostituisce codice e dati di un processo con quelli di un programma differente.

Questo schema mostra la logica di utilizzo congiunto di fork ed exec:



System call exec : sintassi

La exec ha diverse varianti che si differenziano in base a:

- formato degli argomenti (lista o array argv[])
- utilizzo o meno del path della shell

NB: per convenzione il primo argomento è il nome del programma da eseguire

Varianti:

```
execl("/bin/<programma>", arg0, arg1, ..., NULL);
execlp("<programma>", arg0, arg1, ..., NULL);
execv("/bin/<programma>", argv);
execvp("<programma>", argv);
```

Le prime due varianti prendono una lista di argomenti terminata da NULL. Le altre due, invece, prendono i parametri sotto forma di un array di puntatori a stringhe, sempre terminato da NULL. La **presenza della 'p' nel nome della exec** indica che viene utilizzato il path della shell (quindi, ad esempio, non è necessario specificare /bin perché già nel path).

forkexec.c

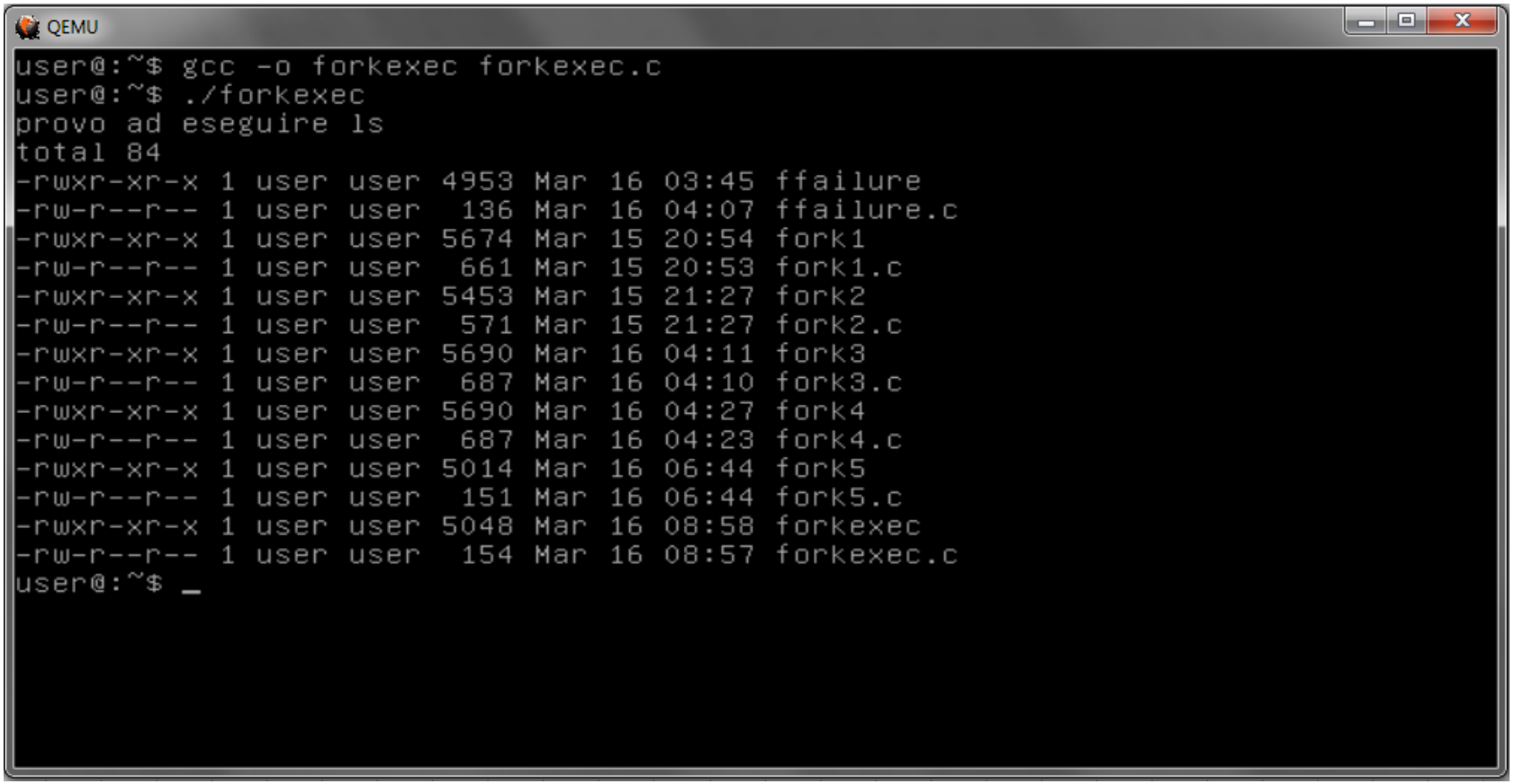
```
#include<stdio.h>
#include <unistd.h>
main() {
      printf("provo a eseguire ls\n");

      execl("/bin/ls","ls","-l",NULL);
      // oppure : execlp("ls","ls","-l",NULL);

      printf("non scrivo questo! \n");
      // questa printf non viene eseguita, se la exec va a buon fine
}
```

Info utili:

\ alt+92

alt+35

La exec ritorna solamente in caso di errore (valore -1). In caso di successo il vecchio codice è completamente sostituito dal nuovo e non è più possibile tornare al programma originale. È estremamente importante capire questo punto.



Come si può vedere, se tutto va a buon fine la exec non ritorna.

THREAD POSIX

Un thread è una unità di esecuzione all'interno di un processo. Un processo può avere più thread in esecuzione, che tipicamente condividono le **risorse del processo** e, in particolare, la **memoria**. Lo standard POSIX definisce un'insieme di funzioni per la creazione e la sincronizzazione di thread. Partiamo da quella utilizzata per la **creazione** di un thread.

pthread_create(pthread_t *thead, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)

- **thread :** un puntatore a **pthread_t**, l'analogo di pid_t. Attenzione che non necessariamente è implementato come un intero.
- **attr :** attributi del nuovo thread. Se non si vogliono modificare gli attributi è sufficiente passare NULL (vedere pthread_attr_init per maggiori dettagli);
- **start_routine :** il codice da eseguire. È un puntatore a funzione che prende un puntatore a void e restituisce un puntatore a void. Ricordarsi che in C il nome di una funzione è un puntatore alla funzione;
- **arg :** eventuali argomenti da passare, NULL se non si intende passare parametri.

THREAD POSIX

Per **terminare** l'esecuzione di un thread si utilizza la funzione pthread_exit che termina l'esecuzione di un thread restituendo un valore di ritorno.

```
pthread_exit(void *retval)
```

Si noti che quando il processo termina (exit) **tutti** i suoi thread vengono terminati. Per far terminare un **singolo** thread si deve usare pthread_exit;

THREAD POSIX

Per **attende la terminazione** di un generico thread th si usa la funzione pthread_join. Se ha successo, ritorna 0 e un puntatore al valore ritornato dal thread.

```
pthread_join(pthread_t th, void **thread_return)
```

Se non si vuole ricevere il valore di ritorno è sufficiente passare **NULL** come secondo parametro

THREAD POSIX

se non si vuole attendere la terminazione di un thread allora si deve eseguire la funzione pthread_detach che pone th in stato detached: **nessun** altro thread potrà attendere la sua terminazione con pthread_join e quando terminerà le sue risorse verranno automaticamente rilasciate (evita che diventino thread "zombie").

```
pthread_detach(pthread_t th)
```

Si noti che pthread_detach **non** fa si che il thread rimanga attivo quando il processo termina con exit.

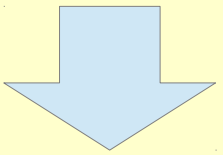
THREAD POSIX

`pthread_t pthread_self()` ritorna il proprio thread id.

ATTENZIONE: questo ID dipende dall'implementazione ed è l'ID della libreria phread e non l'ID di sistema. Per visualizzare l'ID di sistema (quello che si osserva con il comando ps -L, dove L sta per Lightweight process, ovvero thread) si può usare una **syscall specifica** di Linux **syscall(SYS_gettid)**.

threadtest.c (1/2)

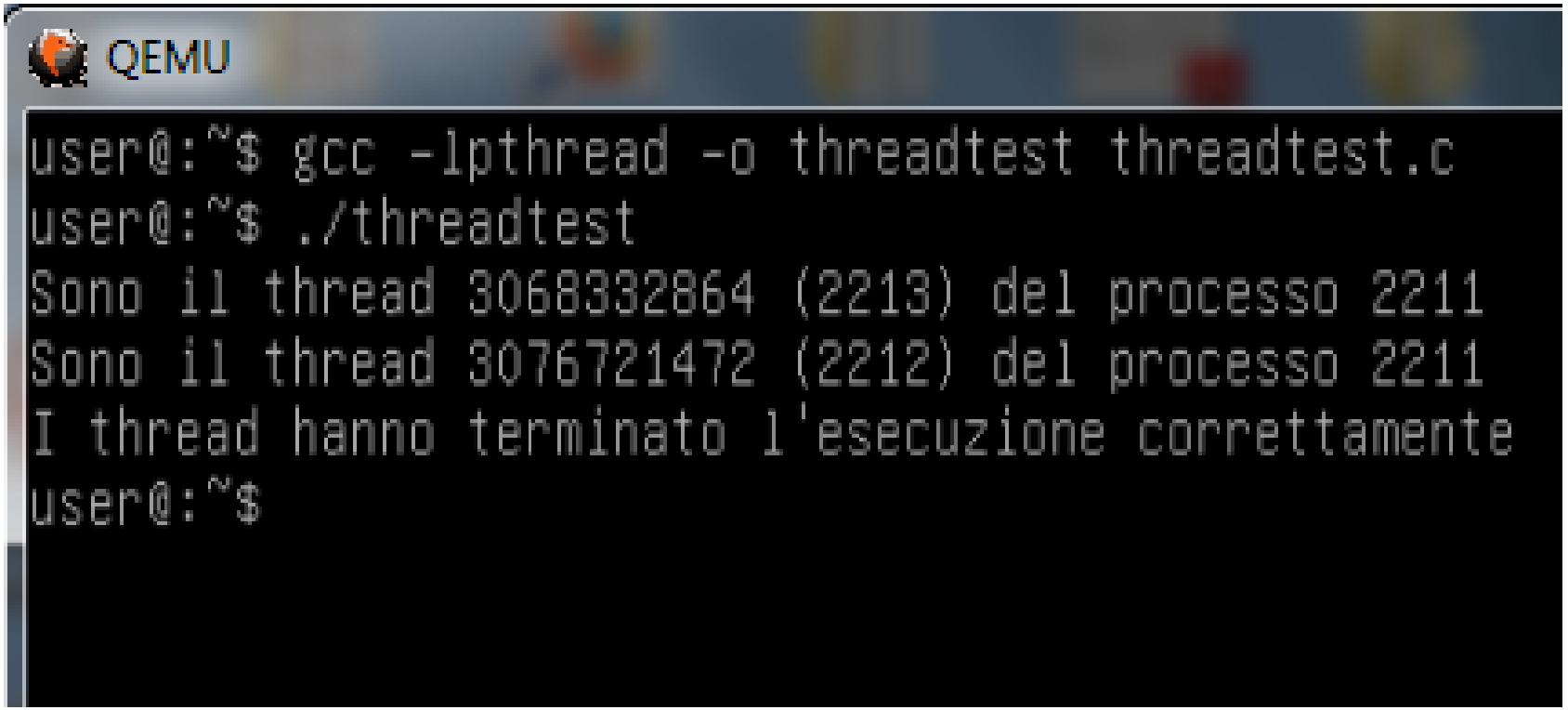
```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>

die(char * s, int e) {                  // stampa gli errori ed esce (non si puo' usare perror)
    printf("%s [%i]\n",s,e);
    exit(1);
}

void * codice_thread(void * a) {     // prende un puntatore e ritorna un puntatore (a void)
    pthread_t tid;
    int ptid;

    tid=pthread_self();                 // library tid
    ptid = syscall(SYS_gettid);         // tid assegnato dal SO

    printf("Sono il thread %lu (%i) del processo %i\n",tid,ptid,getpid());
    sleep(1);
    pthread_exit(NULL);
}
```



threadtest.c (2/2)

```
main() {
    pthread_t tid[2];
    int i,err;

    // crea i thread
    // - gli attributi sono quelli di default (il secondo parametro e' NULL)
    // - codice_thread e' il nome della funzione da eseguire
    // - non vegnono passati parametri (quarto parametro e' NULL)
    for (i=0;i<2;i++)
        if (err=pthread_create(&tid[i],NULL,codice_thread,NULL))
            die("errore create",err);

    // attende i thread. Non si legge il valore di ritorno (secondo parametro NULL)
    for (i=0;i<2;i++)
        if (err=pthread_join(tid[i],NULL))
            die("errore join",err);

    printf("I thread hanno terminato l'esecuzione correttamente\n");
}
```

Compilazione: gcc -lpthread -o threadtest threadtest.c
Esecuzione: ./threadtest


QEMU
user@:~$ gcc -lpthread -o threadtest threadtest.c
user@:~$ ./threadtest
Sono il thread 3068332864 (2213) del processo 2211
Sono il thread 3076721472 (2212) del processo 2211
I thread hanno terminato l'esecuzione correttamente
user@:~$

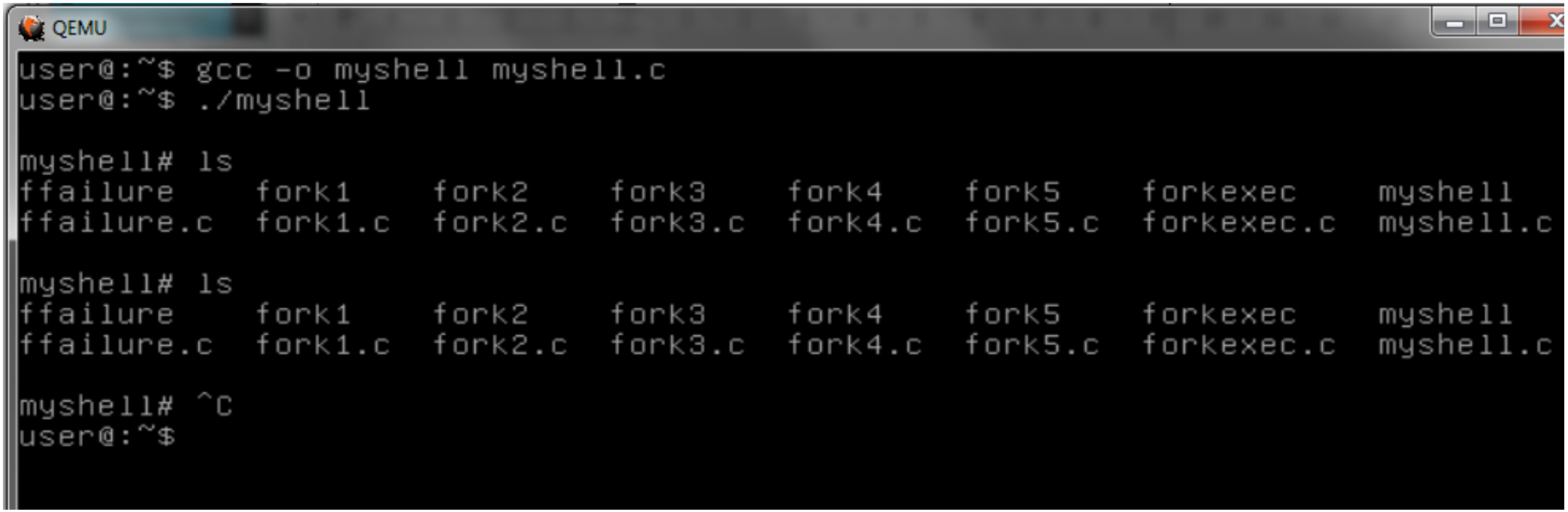
L3 E1

Obiettivo: Simulare una shell. Una shell molto semplice …
Utilizzo di fork, funzioni famiglia exec, wait.
Utilizzo di printf e scanf.

simulare una shell (myshell.c)

```
#include <stdio.h>
 #include <unistd.h>
 #include <stdlib.h>
 main() {
     pid_t esito;
     int status;
     char comando[128];
     while(1) {
         printf("myshell# ");
         scanf("%s", comando); //lettura rudimentale: niente argomenti separati

         if ((esito=fork()) < 0)
             perror("fallimento fork");
         else if (esito == 0) {
             execlp(comando,comando,NULL); // NOTA: non gestisce argomenti
             perror("Errore esecuzione:");
             exit(EXIT_FAILURE);
             }
             else{                                   // codice genitore
               while( wait(&status) != esito );      // aspetta completamento
             }
         // il processo genitore (shell) torna immediatamente a leggere un altro comando
     }
 }
```

L3 E1 : simulare una shell
(myshell.c)



NB: per uscire dalla shell simulata, **ctrl+c**

Provate a utilizzare il comando **top** nella shell simulata.

Comunicazione tra processi : SEGNALI

I segnali sono una forma semplice di comunicazione tra processi: tecnicamente sono **interruzioni software** causati da svariati eventi:

- Generati da terminale. Ad esempio il classico ctrl-c (SIGINT).
- Eccezioni dovute ad errori in esecuzione: es. divisione per 0.
- segnali esplicitamente inviati da un processo all'altro.
- eventi asincroni che vengono notificati ai processi: esempio SIGALARM.

Cosa si può fare quando arriva un segnale?

1) Ignorarlo

2) Gestirlo

3) Lasciare il compito al gestore di sistema

Alcuni segnali POSIX

```
Signal     Value     Action   Comment
-------------------------------------------------------------------
SIGHUP       1       Term    Hangup detected on controlling terminal
                             or death of controlling process
SIGINT       2       Term    Interrupt from keyboard     <== ctrl-C
SIGQUIT      3       Core    Quit from keyboard          <== ctrl-\
SIGILL       4       Core    Illegal Instruction
SIGABRT      6       Core    Abort signal from abort(3)
SIGFPE       8       Core    Floating point exception
SIGKILL      9       Term    Kill signal                 <== kill -9 (da
shell)
SIGSEGV     11       Core    Invalid memory reference
SIGPIPE     13       Term    Broken pipe: write to pipe with no readers
SIGALRM     14       Term    Timer signal from alarm(2)
SIGTERM     15       Term    Termination signal          <== kill (da shell)
SIGUSR1   30,10,16   Term    User-defined signal 1
SIGUSR2   31,12,17   Term    User-defined signal 2
SIGCHLD   20,17,18   Ign     Child stopped or terminated <== gestito da
wait()
SIGCONT   19,18,25   Cont    Continue if stopped
SIGSTOP   17,19,23   Stop    Stop process
```

un esempio semplice (alarmtest.c)

```
main(){

    alarm(3);
    While(1){};    // questo causerebbe un ciclio infinito
}
```

Compilazione: gcc -o alarmtest alarmtest.c
Esecuzione: ./alarmtest

Output:

linuxprompt$ ./alarmtest
Alarm clock

dopo 3 secondi ...

Il programma "punta una sveglia" dopo 3 secondi. Allo scadere dei 3 secondi viene inviato un segnale **SIGALRM** al processo. Il comportamento di default è quello di terminare il processo.

Impostare il gestore dei segnali mediante SIGNAL

(alarmhandler.c)

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

void alarmHandler()
{
    printf("questo segnale lo gestisco io!\n");
    alarm(3); // resetta il timer a 3 secondi
}

main() {
    signal(SIGALRM, alarmHandler);
    alarm(3);
    while(1){}
}
```

Compilazione: gcc -o alarmhandler alarmtest.c
Esecuzione: ./alarmhandler

Output: Ogni 3 secondi scrive “questo segnale lo gestisco io” sullo schermo.

Un altro esempio: proteggersi da ctrl+c

(noctrlc.c)

```
#include<signal.h>
#include<stdio.h>
main() {
    void (*old)(int);

    old = signal(SIGINT,SIG_IGN);
    printf("Sono protetto!\n");
    sleep(3);

    signal(SIGINT,old);
    printf("Non sono più protetto!\n");
    sleep(3);
}
```

Se modifichiamo il gestore del segnale SIGINT possiamo evitare che un programma venga interrotto tramite **ctrl-c** da terminale.

NB: Il **valore di ritorno** della signal viene usato per reimpostare il gestore originale. La signal, quando va a buon fine, ritorna il gestore precedente del segnale, che salviamo nella variabile old. Quando vogliamo reimpostare tale gestore è sufficiente passare old come secondo parametro a signal.

Un altro esempio: proteggersi da ctrl+c

Compilazione: gcc -o noctrlc noctrlc.c
Esecuzione: ./noctrlc

Output:

```
$ ./noctrlc
Sono protetto!
^C^C^CNon sono più protetto!
^C
$
```

Premete ctrl+c … qui non ha effetto

Qui ctrl+c causa l'interruzione dell'esecuzione

Comunicazione tra processi

In un sistema operativo ci sono un numero molto elevato di programmi in esecuzione (processi). Idealmente tali processi dovrebbero "comportarsi bene" evitando di interferire uno con l'altro. Nella pratica, difficilmente il comportamento di un processo è indipendente da quello degli altri processi in esecuzione.

Competizione

I processi competono innanzitutto per le risorse comuni. Ad esempio:

- Apertura dello stesso file
- Utilizzo della stessa stampante
- Condivisione della CPU

La competizione per le risorse **crea interferenze** tra processi.

Comunicazione tra processi: esempi di interferenze

Starvation: un processo è bloccato indefinitamente a causa di altri processi che monopolizzano una o più risorse. Abbiamo visto un esempio estremo (slide 23, esempio di fork-bomb)

I/O: L'accesso a una risorsa di input/output può variare notevolmente di prestazioni a seconda di quanti processi la stanno utilizzando.

Il sistema operativo deve **gestire la competizione su risorse comuni** in modo da ridurre il più possibile le interferenze e da garantire correttezza.

In definitiva l'interazione tra processi non sempre è un evento positivo ed il s.o. deve gestire tutto in modo da evitare inconvenienti. Tuttavia, a volte, l'interazione tra processi è non solo utile ma indispensabile per la realizzazione di alcuni task.

Esempi di interazione “positiva”

- **Condivisione:** quando si vuole condividere informazione è necessario interagire. Ad esempio in un progetto software o un wiki. Ci sono molti file nel sistema che sono condivisi da diverse applicazioni

- **Prestazioni:** con le architetture multi-core si possono utilizzare algoritmi paralleli per aumentare le prestazioni. Se un programma è scritto in modo sequenziale non andrà a sfruttare la presenza di più core

- **Modularità:** un’applicazione complessa viene spesso suddivisa in attività distinte più semplici, ognuna delle quali viene eseguita da un programma distinto (processo o thread). Un correttore ortografico in un editor di testo o in un browser è un tipico esempio: la ricerca di errori avviene parallelamente all’attività principale ma chiaramente i dati (il testo) sono condivisi ed è necessaria una interazione.

Esempi di interazione “positiva”

I comandi UNIX forniscono un buon esempio di interazione positiva che sfrutta in modo efficace il concetto di modularità.

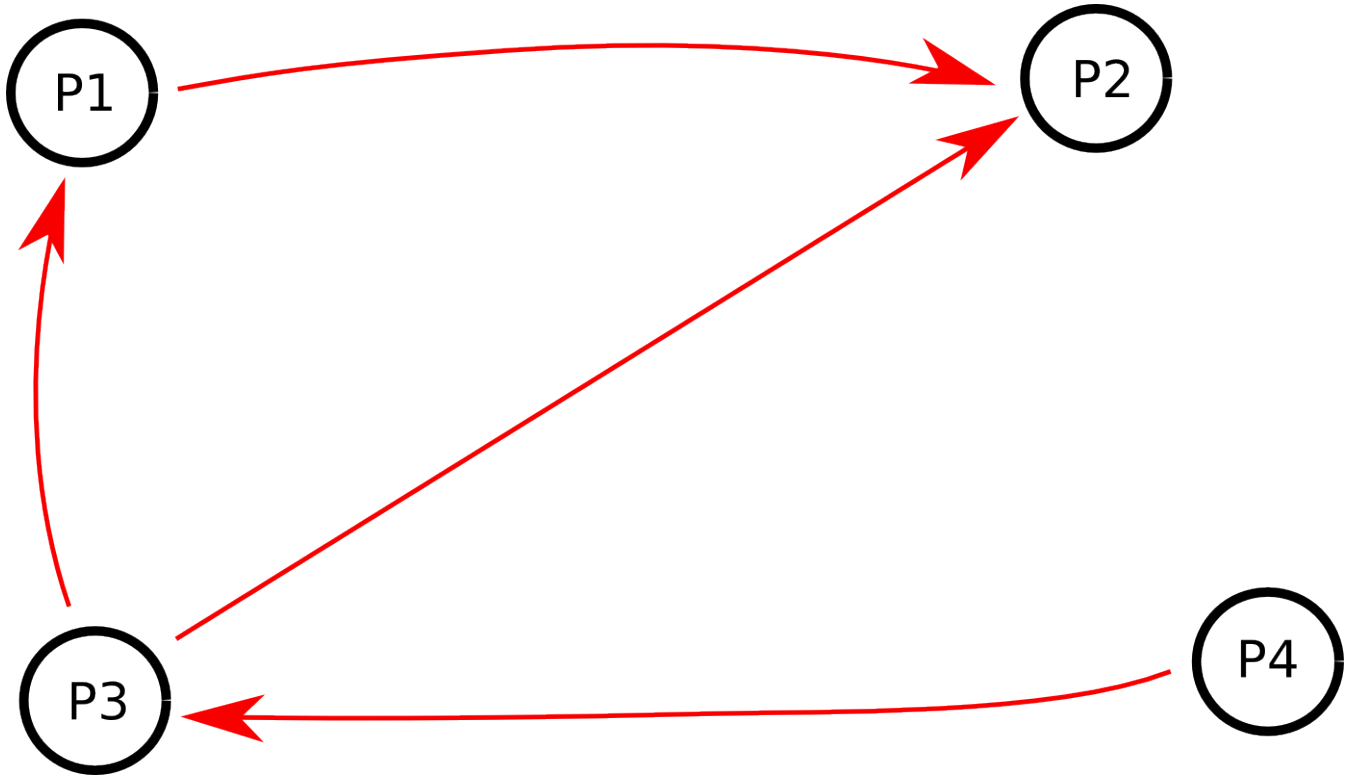
```
$ ls -al | grep fork
-rw-rw-r--   1 user user  fork1.c
-rw-r--r--   1 user user   fork2.c
$
```

Il comando ls -al mostra il contenuto del folder. Il suo **output viene dato in input a un secondo comando** ad es. grep pippo che stampa solo le righe contenenti la parola pippo. Il simbolo “|” (pipe) indica appunto che l’output del primo comando deve essere dato in input al secondo. In questo modo otteniamo un comportamento utile combinando due comandi più semplici.

Modelli di comunicazione

Per cooperare è necessario comunicare. Esistono due modelli fondamentali di comunicazione tra processi e thread.

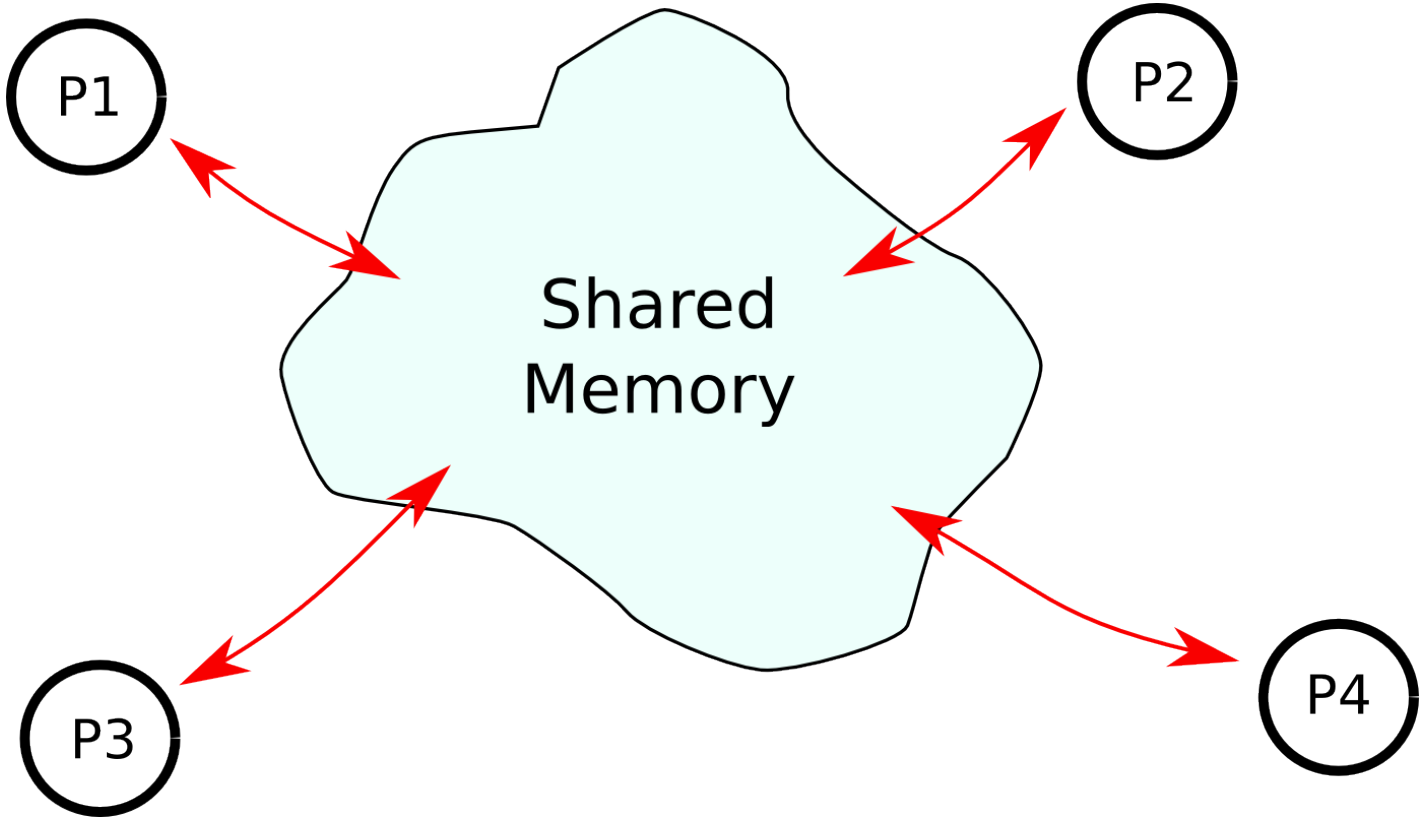
Message passing (scambio di messaggi): i processi o thread si scambiano informazioni tramite messaggi, un po' come avviene sulla rete



Modelli di comunicazione

Per cooperare è necessario comunicare. Esistono due modelli fondamentali di comunicazione tra processi e thread.

Shared memory (memoria condivisa): i processi o thread condividono dati in memoria e accedono in lettura e scrittura a tali dati condivisi.



Scambio di messaggi

I processi dispongono di due primitive

- **send**(m), invia il messaggio m;
- **receive**(&m), riceve un messaggio e lo salva in m.

Vengono realizzate tramite opportune System Call dette InterProcess Communication (IPC).

Mittente e destinatario possono essere indicati direttamente o indirettamente.

Nominazione diretta: mittente e destinatario sono nominati esplicitamente

send(P,m)	invia il messaggio m a P
receive(Q,&m)	riceve un messaggio da Q e lo salva in m

Canale riservato per ogni coppia di processi. Basta conoscere la reciproca identità. Esiste anche una variante in cui si riceve da qualsiasi processo

receive(&Q,&m)	riceve un messaggio m da un qualsiasi processo messaggio e mittente vengono salvati in m e Q

Vantaggi: la nominazione diretta è molto semplice e permette di comunicare in modo diretto tra coppie di processi.
Svantaggi: È però necessario un “accordo” sui nomi dei processi. Il PID infatti viene dato dinamicamente dal sistema e non possiamo prevederne il valore a priori. Nella pratica è difficile da implementare a meno che i processi non siano in relazione stretta di parentela genitore-figlio.

Nominazione indiretta:

Per ovviare ai difetti della nominazione diretta si utilizza, in pratica, una nominazione indiretta tramite **porte**. Le porte sono gestite dal sistema operativo tramite opportune chiamate che ne permettono la creazione e distruzione.

send(A,m), invia il messaggio m sulla porta A;
receive(A,&m), riceve un messaggio dalla porta A e lo salva in m.

In questo modo **non è necessario conoscere il nome dei processi** ma solamente quello delle porte.

le **pipe di Unix** implementano un meccanismo di scambio di messaggi a nominazione indiretta. Il riferimento alla pipe può avvenire tramite un descrittore (pipe senza nome) oppure tramite un nome conosciuto a livello di file-system (pipe con nome).

Comunicazione tra processi

La comunicazione a scambio di messaggi è molto adatta nelle situazione in cui un processo “produce” un dato e un altro lo “consuma”. Abbiamo visto l’esempio Unix:

linuxprompt$ ls -al | grep fork

Il processo ls -al produce un output che viene dato in input a grep fork che lo “consuma” generando un ulteriore output.

Pipe

Le pipe sono la forma più “antica” di comunicazione tra processi UNIX. Una pipe, che letteralmente significa **tubo**, costituisce un canale di comunicazione tra due processi: si possono **inviare** dati da un lato della pipe e **riceverli** dal lato opposto. Tecnicamente, la pipe è una porta di comunicazione (nominazione indiretta).

Esistono due forme di pipe in UNIX: **senza nome** e **con nome**. Le prime sono utilizzabili solo da processi con antenati comuni, in quanto sono risorse che vengono **ereditate dai genitori**. Le seconde, invece, hanno un **nome nel filesystem** e costituiscono quindi delle porte che tutti i processi possono utilizzare.

Pipe senza nome

Le pipe senza nome sono utilizzate per combinare comandi Unix direttamente dalla shell tramite il simbolo “|” (pipe).

Per creare una pipe si utilizza la systemcall **pipe(int filedes[2])** che restituisce in filedes due descrittori di file:

filedes[0]	per la **lettura**
filedes[1]	per la **scrittura**

Le pipe, quindi, sono half-duplex (monodirezionali): esistono due distinti descrittori per leggere e scrivere. Per il resto, una pipe si utilizza come un normale file come mostra l'esempio seguente.

Un momento … prima dell'esempio **vediamo alcune syscall per la gestione dei file**.

FILE

Un file è un insieme di byte conservati nella memoria di massa. Hanno associato un **nome** ed altri attributi. Nei sistemi unix-like sono organizzati gerarchicamente in directory (l'equivalente dei folder in MS Windows), che non sono che altri file contenenti un elenco...

POSIX syscall (file mgt)

fd = creat(name, mode)	Obsolete way to create a new file
fd = mknod(name, mode, addr)	Create a regular, special, or directory i-node
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
pos = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information
s = fstat(fd, &buf)	Get a file's status information
fd = dup(fd)	Allocate a new file descriptor for an open file
s = pipe(&fd[0])	Create a pipe
s = ioctl(fd, request, argp)	Perform special operations on a file
s = access(name, amode)	Check a file's accessibility
s = rename(old, new)	Give a file a new name
s = fcntl(fd, cmd, ...)	File locking and other operations

POSIX syscall (file mgt)

s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system
s = sync()	Flush all cached blocks to the disk
s = chdir(dirname)	Change the working directory
s = chroot(dirname)	Change the root directory

Ora che siamo attrezzati con le syscall per la gestione dei file (in particolare read/write e open/close) possiamo procedere con l'esempio sulle pipe.

Sistemi Operativi

Laboratorio – linea 2

3

(testpipe.c)

```
#include <stdio.h>
#include <string.h>
main() {
        int fd[2];

        pipe(fd); /* crea la pipe */
        if (fork() == 0) {
             char *phrase = "prova a inviare questo!";

             close(fd[0]);                              /* chiude in lettura */
             write(fd[1],phrase,strlen(phrase)+1); /* invia anche 0x00 */
             close(fd[1]);                              /* chiude in scrittura */
        } else {
             char message[100];
             memset(message,0,100);
             int bytesread;

             close(fd[1]);                              /* chiude in scrittura */
             bytesread = read(fd[0],message,100);
             printf("ho letto dalla pipe %d bytes: '%s' \n",bytesread,message);
             close(fd[0]);                              /* chiude in lettura */
        }
}
```

Pipe

Compilazione: gcc -o testpipe testpipe.c
Esecuzione: ./testpipe

Output:

ho letto dalla pipe 24 bytes: 'prova a inviare questo!'

Pipe e comandi UNIX

Abbiamo visto come le pipe permettano di eseguire comandi UNIX (nella shell) in serie fornendo l'output prodotto da un programma in input al comando successivo.

Questo comportamento permette di effettuare operazioni estremamente complesse. Noi abbiamo visto solo un semplice esempio in cui era presente un'unica pipe. Ma provate a scrivere questa serie di comandi nella shell:

```
cat /usr/include/string.h | xargs -n1 | tr A-Z a-z | sed -e 's/\.//g' -e 's/\,//g' -e 's/ //g' | sort | uniq -c | sort -nr
```

Elenca il file una parola per riga
Cambia tutte le lettere maiuscole in minuscole
Filtra I punti e le virgole. Cambia gli spazi tra parole in linefeed
Conteggio occorrenze e ordinamento in base al numero

DOMANDA: qual'è la parola più frequente in string.h ?

Lanciare comandi con la shell (quella vera ...)

- Per iniziare l'esecuzione di un programma basta scrivere il nome del file
  - /bin/ls
- Il programma è trattato come una *funzione*, che prende dei parametri e ritorna un intero (`int main(int argc, char*argv[])`). Convenzione: 0 significa "non ci sono stati errori", >0 errori (2 errore nei parametri), parametri - $\rightsquigarrow$ opzioni
  - /bin/ls /usr
  - /bin/ls piripacchio
- Si può evitare che il padre aspetti la terminazione del figlio
  - /bin/ls /usr &
- Due programmi in sequenza
  - /bin/ls /usr ; /bin/ls /usr
- Due programmi in parallelo
  - /bin/ls /usr & /bin/ls /usr

esecuzione in background

Sistemi Operativi

Laboratorio – linea 2

3

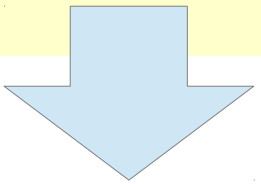
ESERCIZI

1. Scrivere, compilare (`cc -o nome nome.c`) ed eseguire un programma che *forca* un nuovo processo.
2. Scrivere un programma che stampi sullo schermo ``‘‘Hello world! (numero)’’`` per 10 volte alla distanza di 1 secondo l'una dall'altra (`sleep(int)`). Terminare il programma con una chiamata `exit(0)`
3. Usare il programma precedente per sperimentare l'esecuzione in sequenza e in parallelo
4. Controllare il valore di ritorno con `/bin/echo $?`
5. Tradurre il programma in assembly con `cc -S nome.c`
6. Modificare l'assembly affinché il programmi esca con valore di ritorno 3 e controllare con `echo $?` dopo aver compilato con `cc -o nome nome.s`
7. Modificare l'assembly in modo che usi `scanf` per ottenere il numero di saluti.

Prima di affrontare esercizio 7 …

Leggete attentamente questo sorgente assembly.

```
; add2int.asm
SECTION .data
    message1: db "Enter the first number: ", 0
    message2: db "Enter the second number: ", 0
    formatin: db "%d", 0
    formatout: db "%d", 10, 0 ; newline, nul terminator
    integer1: times 4 db 0 ; 32-bits integer = 4 bytes
    integer2: times 4 db 0 ;
SECTION .text
   global _main
   extern _scanf
   extern _printf
```



(continua...)

Sistemi Operativi

Laboratorio – linea 2

3

Prima di affrontare esercizio 7 …

```
_main:

   push ebx ; save registers
   push ecx
   push message1
   call printf
   add esp, 4 ; remove parameters
   push integer1 ; address of integer1 (second parameter)
   push formatin ; arguments are right to left (first parameter)
   call scanf

   add esp, 8 ; remove parameters
   push message2
   call printf

   add esp, 4 ; remove parameters
   push integer2 ; address of integer2
   push formatin ; arguments are right to left
   call scanf

   add esp, 8 ; remove parameters
```

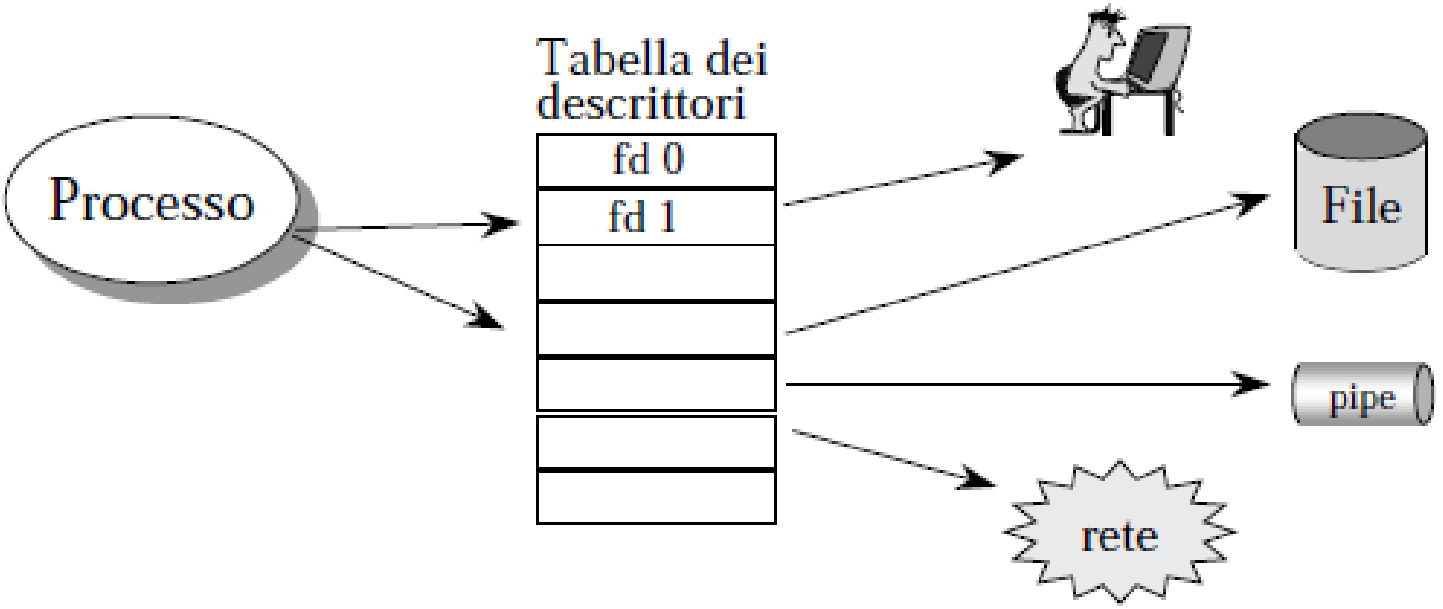
DOMANDA: la compilazione/link richiedono qualcosa di particolare?

File in UNIX

Un processo Unix vede tutto il mondo esterno (I/O) come un insieme di descrittori (da qui discende omogeneità tra file e dispositivi di Unix).

I file descriptor sono piccoli interi non negativi che identificano i file aperti **standard input**, **standard output**, **standard error** sono associati ai file descriptor 0, 1, 2.

Nuove operazioni di RICHIESTA producono nuovi file descriptor per un processo.



File in UNIX

I processi interagiscono con l'I/O secondo il paradigma open-read-write-close (operazioni di **prologo** e di **epilogo**)

Flessibilità (possibilità di pipe e ridirezione)

System Call per operare a basso livello sui file : (**creat**, **open**, **close**, **read/write**, **lseek**).

File in UNIX

CREATE

fd = **creat**(name,mode);

```
int fd;          /* file descriptor  */
int mode;        /* attributi del file */
```

⇨ diritti di UNIX (di solito espressi in ottale)

⇨ file name aperto in scrittura

OPEN

fd = **open**(name, flag);

```
char *name;
int flag;   /* 0 lettura, 1 scrittura, 2 entrambe */
int fd;     /* file descriptor  */
```

⇨ apre il file di nome **name** con modalità **flag**

⇨ in **/usr/include/fcntl.h** sono definite le costanti O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_TRUNC, O_EXCL

PROLOGO
(apertura/creazione)

Esempi

```
fd=open("file", O_WRONLY| O_APPEND)
fd=open("file", O_WRONLY| O_CREAT| O_APPEND, 0644)
fd=open("file", O_WRONLY| O_CREAT| O_TRUNC, 0644)
fd=open("lock", O_WRONLY| O_CREAT| O_EXCL, 0644)
```

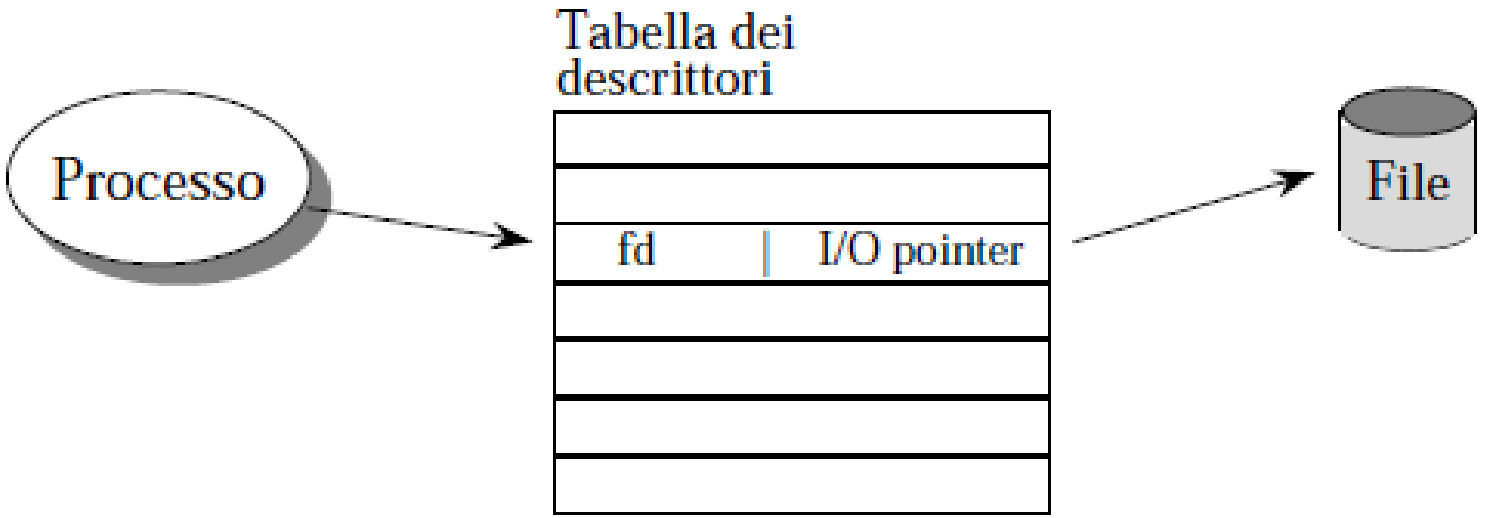
File in UNIX

EPILOGO (chiusura)

CLOSE

```
retval = close(fd);
int fd, retval;
```

I file in Unix sono una sequenza di **byte**. Accesso **sequenziale**. I file sono rappresentati dai file descriptor. Presenza di I/O pointer associato al file (e al processo). I/O pointer punta alla **posizione corrente** del file su cui il processo sta operando (scrivendo/leggendo).



File descriptor

In generale,

la lettura da fd 0 → legge da standard input
la scrittura su fd 1 → scrive su standard output
la scrittura su fd 2 → scrive su standard error

Questi tre file descriptor sono aperti automaticamente dal sistema (shell) per ogni processo e collegati all'I/O

Per progettare FILTRI cioè usare RIDIREZIONE e PIPING

I filtri leggono direttamente dal file descriptor **0** , scrivono direttamente sul file descriptor **1**

Operazioni di lettura e scrittura

READ nread = **read**(fd, buf, n);
WRITE nwrite = **write**(fd, buf, n);

```
int nread, nwrite, n, fd
char *buf
```

- **Lettura e scrittura** di un file avvengono a partire dalla posizione corrente del file ed avanzano il puntatore (I/O pointer) all'interno del file
- **Restituiscono:** il numero dei byte su cui hanno lavorato, -1 in caso di errore

Meccanismi di protezione

Ogni utente ha la **propria visione** dei file aperti:

- Nel caso di più utenti che aprono lo stesso file, ogni processo utente ha un proprio I/O pointer separato
- SE un utente legge o scrive, modifica solo il proprio pointer, non modifica l'I/O pointer di altri!

La system call lseek

La system call lseek permette l'accesso random ad un file, cambiando il numero del prossimo byte da leggere/scrivere.

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

off_t lseek(int filedes, off_t offset, int start_flag);
```

Il parametro **filedes** è un descrittore di file. Il parametro **offset** determina la nuova posizione del puntatore in lettura/scrittura. Il parametro **start_flag** specifica da dove deve essere calcolato l'offset. Startflag può assumere uno dei seguenti valori simbolici:
SEEK SET (0) : offset è misurato dall'inizio del file
SEEK CUR (1) : offset è misurato dalla posizione corrente del puntatore
SEEK END (2) : offset è misurato dalla fine del file

lseek ritorna la **nuova posizione del puntatore**.

L3

esperimenti file

FILE (testfile.c) 1/2

```
1  main(){
2          pid_t pid;
3          int f, off;
4          char string[] = "Hello, world!\n";
5
6          lsofd("padre (senza figli)");
7          printf("padre (senza figli) pipe *\n");
8          f = open("provaxxx.dat", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
9          if (f == -1){
10                 perror("pipe");
11                 exit(1);
12         }
13         lsofd("padre (senza figli)");
14         if (write(f, string, (strlen(string)+1)) != (strlen(string)+1) ){
15                 perror("write");
16                 exit(1);
17         }
18
19         off = lseek(f, 0, SEEK_CUR);
20         printf("padre (senza figli) seek: %d\n", off);
21
22         printf("padre (senza figli) fork *\n");
23         if ( (pid = fork()) < 0){
24                 perror("fork");
25                 exit(1);
26         }
```


```
1         if (pid > 0){
2                 lsofd("padre");
3                 printf("padre close *\n");
4                 printf("padre write *\n");
5                 off = lseek(f, 0, SEEK_CUR);
6                 printf("padre seek prima: %d\n", off);
7                 if (write(f, string, (strlen(string))) != (strlen(string)) ){
8                         perror("write");
9                         exit(1);
10                }
11                lsofd("padre");
12                off = lseek(f, 0, SEEK_CUR);
13                printf("padre seek dopo: %d\n", off);
14                exit(0);
15        }
16        else {
17                lsofd("figlio");
18                printf("figlio close *\n");
19                printf("figlio write *\n");
20                off = lseek(f, 0, SEEK_CUR);
21                printf("figlio seek prima: %d\n", off);
22                if (write(f, string, (strlen(string))) != (strlen(string)) ){
23                        perror("write");
24                        exit(1);
25                }
26                lsofd("figlio");
27                off = lseek(f, 0, SEEK_CUR);
28                printf("figlio seek dopo: %d\n", off);
29                exit(0);
30        }
31 }
```

FILE (testfile.c) 1/2

Per sperimentare con I file descriptor può essere utile una funzione come questa

```
1  #include <stdio.h>
2  #include <sys/stat.h>
3  #define _POSIX_SOURCE
4  #include <limits.h>
5
6  void lsofd(void){
7          int i;
8          for (i=0; i<_POSIX_OPEN_MAX; i++){
9                  struct stat buf;
10                 if (fstat(i, &buf) == 0){
11                         printf("fd:%d i-node: %d\n", i, buf.st_ino);
12                 }
13         }
14 }
```