Creazione di processi

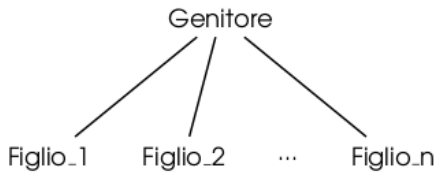
La creazione di un processo richiede alcune operazioni da parte del Sistema Operativo:

- 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. priorita),
- Creazione PCB (Process Control Block) contenente le informazioni precedenti.

Processi in Unix

Un processo è sempre creato da un altro processo tramite un'opportuna chiamata a sistema. Fa eccezione `init` (pid = 1) che viene creato al momento del boot.

Il processo creante e detto **parent** (genitore), mentre il processo creato **child** (figlio). Si genera una struttura di "parentela" ad albero.



Relazioni Dinamiche

Cosa accade dopo la creazione?

- Il processo genitore attende il processo figlio.
  Esempio: Esecuzione di un programma da shell

```
> xcalc
<la shell attende la terminazione del programma>
```

  La shell è genitore del nuovo processo `xcalc`
  NOTA: il terminale e associato al nuovo programma: `ctrl-c`, ad esempio, termina la calcolatrice.

- Il processo genitore continua
  esempio: Esecuzione in *background* di un programma da shell:

```
> xcalc &
[1] 17589    <-- PID del processo figlio
> ps
 PID TTY          TIME CMD
14695 pts/1    00:00:00 bash
17589 pts/1    00:00:00 xcalc
17590 pts/1    00:00:00 ps
>
```

  In questo caso i due processi procedono in modo *concorrente*. Tramite `ps` possiamo osservare i processi in esecuzione.

  NOTA: `ps` di default mostra solo i processi associati al terminale da cui viene lanciato. In questo caso abbiamo la shell `bash`, il programma `xcalc` e lo stesso `ps`.

  Anche in esecuzione rimane un legame genitore-figlio. Ad esempio, se chiudiamo la calcolatrice la shell viene notificata:

```
>
[1]+  Done                    xcalc
>
```

  Può essere utile che un processo si dissoci dal processo genitore. Ad esempio i *Daemon* sono processi che restano attivi fino allo shutdown del sistema: devono, ad esempio, dissociarsi dalla shell per non essere terminati alla chiusura del terminale stesso. Quando si dissociano diventano figli del processo `init` (vedremo come questo sia possibile).

Per visualizzare il PID del genitore basta passare gli opportuni parametri al programma ps (PPID significa Parent process ID):

Per visualizzare il PID del genitore basta passare gli opportuni parametri al programma ps (PPID significa Parent process ID):

```
> ps -o pid,ppid,command
  PID  PPID COMMAND
14695 14201 bash
17679 14695 xcalc
17680 14695 ps -o pid,ppid,command
>
```

Si osservi che xcalc e ps sono entrambi figli di bash.

Relazioni di contenuto

Due possibilità:

- Il figlio è un duplicato del genitore (ad esempio in UNIX)
- Il figlio esegue un programma differente (ad esempio nei sistemi Windows)

Questo è il comportamento standard ma ovviamente è possibile anche l'altra modalità in entrambi i sistemi.

System Call "fork"

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

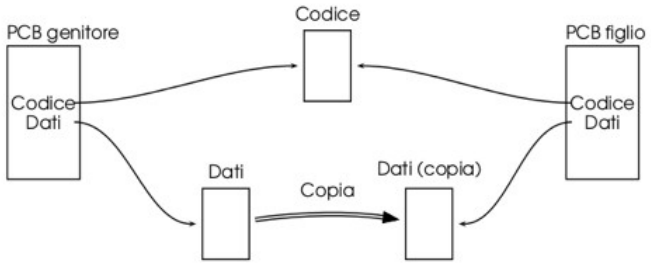
NOTA `fork`, come la maggior parte delle chiamate a sistema che discuteremo, appartiene allo standard POSIX (Portable Operating System Interface) di IEEE (Institute of Electrical and Electronics Engineers). È utilizzabile in qualsiasi sistema che supporti POSIX.

La `fork` crea un nuovo processo che

- condivide l'area codice del processo genitore
- utilizza una copia dell'area dati del processo genitore

Come fanno i processi a differenziarsi? Si utilizza il valore di ritorno della `fork`:

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



Schema di base di utilizzo della fork:

```
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: da usare con cautela!
```

NOTA: `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`

Un esempio concreto:

Un esempio concreto:

```
// Esempio di utilizzo della fork.
#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());
}
```

che dà il seguente output:

```
> a.out
Prima della fork. pid = 25267, pid del genitore = 329
[Genitore] pid = 25267, pid del mio genitore = 329
[Genitore] Mio figlio ha pid = 25268
[Figlio] pid = 25268, pid del genitore = 25267
PID 25268 termina.
PID 25267 termina.
>
```

Esempio Come possiamo visualizzare l'alternanza nell'esecuzione dei processi genitore e figlio? Un modo è mettere le printf dentro un `while(true)`:

```
// visualizza l'esecuzione concorrente di genitore e figlio
#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<10000;i++) {} // riduce il numero di printf
                        printf("Figlio: pid = %d, pid del genitore = %d\n",getpid(), getppid());
                }
        } else {
                while(true) {
                        for (i=0;i<10000;i++) {} // riduce il numero di printf
                        printf("genitore: pid = %d, pid di mio genitore = %d\n",getpid(), getppid());
                }
        }
}
```

In output avremo un'alternanza (non stretta a causa dell'output bufferizzato!) degli output dei due processi. Il ciclo for "vuoto" prima della `printf` riduce il numero di stampe e permette di visualizzare meglio l'alternanza (ed evita di "saturare" il terminale di output). Variare, se necessario, il limite.

NOTA Su computer a più core i processi vengono eseguiti in parallelo su core differenti. In tale case non è necessario rallentarli con il ciclo for per vedere l'alternanza nell'output. Per vedere il carico sui vari core si può lanciare 'top' da un altro terminale. Il Linux, premendo 1, viene visualizzato il carico delle differenti CPU. Provare per esercizio.

Fallimento della fork

Quando fallisce una `fork`? Quando non è possibile creare un processo e tipicamente questo accade quando non c'è memoria per il processo o per il kernel. Ecco un piccolo test.

NOTA BENE. Il test potrebbe bloccare tutto il sistema perché i processi generati vanno ad occupare tutte le risorse e il sistema non può più prendere il controllo. È necessario limitare in 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.

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

Esercizio. Quanti processi eseguono la `fork` al loop i-esimo? (pensare a quanti processi ci sono alla prima fork, quanti alla seconda, e cosi' via).

Processi orfani e processi zombie

Se metto una `sleep` subito prima della printf nel figlio lo rendo *orfano* perché termina il genitore prima di lui: viene adottato da init (processo con pid = 1):

```
// figlio
sleep(5);
printf("[Figlio] pid = %d, pid del genitore = %d\n",getpid(), getppid());
```

ottengo:

```
> a.out
Prima della fork. pid = 8127, pid del genitore = 5835
genitore: pid = 8127, pid di mio genitore = 5835
Mio figlio ha pid = 8128
PID 8127 termina.
>                                          <== 5 secondi
Figlio: pid = 8128, pid del genitore = 1        <== processo adottato!
PID 8128 termina.
```

dove si nota, appunto, l'adozione da parte di init.

NOTA Un processo orfano non viene più terminato da ctrl-c o dalla chiusura della shell (provare).

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

```
// genitore
sleep(5);
printf("[genitore] pid = %d, pid di mio genitore = %d\n",getpid(),getppid());
```

che dà il seguente output:

```
> a.out &                       <== notare l'esecuzione in background
[1] 8270
Prima della fork. pid = 8270, pid del genitore = 5835
Figlio: pid = 8271, pid del genitore = 8270
PID 8271 termina.               <== termina figlio e diventa zombie
> ps                            <== andiamo ad osservare i processi
  PID TTY          TIME CMD
 5835 pts/1    00:00:00 bash
 8270 pts/1    00:00:00 a.out                  <== genitore in esecuzione
 8271 pts/1    00:00:00 a.out <defunct>        <== zombie
 8272 pts/1    00:00:00 ps
>                                          <== 5 secondi
 genitore: pid = 8270, pid di mio genitore = 5835
Mio figlio ha pid = 8271
PID 8270 termina.
[1]+  Exit 18                 a.out
```

Esecuzione e terminazione

Esercizio. 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.

Considerare il seguente 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: che output dà? Perchè?

Soluzione: guardarla solo **dopo** aver provato a risolvere l'esercizio da soli!

System call exec

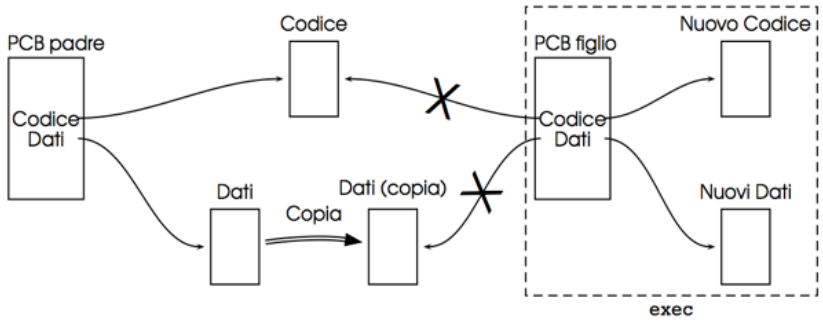
Come si fa ad eseguire un programma diverso da quello che ha effettuato la `fork`? Esiste una chiamata a sistema apposita: `exec`. Tale chiamata a sistema **sostituisce** *codice* e *dati* di un processo con quelli di un programma differente.

Lo schema seguente mostra `fork` ed `exec` assieme.

System call exec

Come si fa ad eseguire un programma diverso da quello che ha effettuato la `fork`? Esiste una chiamata a sistema apposita: `exec`. Tale chiamata a sistema **sostituisce** *codice* e *dati* di un processo con quelli di un programma differente.

Lo schema seguente mostra `fork` ed `exec` assieme.



Copy-on-write

Notare che la `exec` "butta via" la copia dei dati creata dalla `fork`. Questo è chiaramente inefficiente, soprattutto quando la `exec` viene eseguita immediatamente dopo la `fork`. Per ovviare a questo problema, viene copiata solamente la page-table, e le pagine (quelle contenenti i dati, che dovrebbero essere state copiate) sono invece etichettate come *read-only*. Un tentativo di scrittura, quindi, genera un errore che viene gestito dal kernel:

1. copiando al volo (**copy-on-write**, appunto) la pagina fisica e aggiornando opportunamente la page-table in modo che punti alla nuova copia;
2. impostando la modalità a read-write: da quel momento in poi le due copie sono indipendenti.

Quindi se si fa `fork` e subito `exec` non avviene nessuna scrittura e quindi nessuna pagina viene effettivamente copiata.

Nota storica: Precedentemente alla tecnica Copy-on-write veniva utilizzata la `vfork`, che condivideva i dati con il processo genitore in attesa di eseguire la `exec`. Eseguire `man vfork` per maggiori informazioni.

Sintassi

La `exec` ha diverse varianti che si differenziano in base al

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

```
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).

NOTA: Per convenzione, il primo argomento contiene il nome del file associato al programma da eseguire.

Valore di ritorno

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. L'esempio successivo lo evidenzia e permette di fare alcuni test interessanti.

```
#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
}
```

Dà il seguente output

```
provo a eseguire ls
total 16
-rwxr-xr-x 1 focardi focardi 6619 2008-03-05 17:02 a.out
-rw-r--r-- 1 focardi focardi  226 2008-03-05 17:02 exec1.c
-rw-r--r-- 1 focardi focardi  225 2008-03-05 17:02 exec1.c~
```

Si può quindi osservare che in effetti la exec **non ritorna** se il comando viene eseguito correttamente! Se sostituiamo il seguente comando alla exec del programma precedente generiamo un errore, in quanto ls2 non esiste:

```
      execlp("ls2","ls2","-l",NULL);
```

L'esecuzione in questo caso dà il seguente risultato:

```
provo a eseguire ls
non scrivo questo!
```

In questo caso la exec è andata in errore e il controllo ritorna al programma. Di conseguenza l'ultima `printf` viene eseguita stampando la stringa "non scrivo questo!"

È quindi buona norma verificare se la `exec` ritorna -1 e in tal caso gestire l'errore (in generale in C è sempre consigliabile **testare il valore di ritorno** delle chiamate a libreria e a sistema e stampare un messaggio di errore, se necessario, altrimenti diventa complesso capire dove il programma sta fallendo)

```
if (execlp("ls2","ls2","-l",NULL) == -1) {
    perror("errore durante la exec");
    // eventualmente si esce: exit(EXIT_FAILURE);
    }
```

In questo caso otteniamo:

```
provo a eseguire ls
errore durante la exec: No such file or directory
non scrivo questo!
```

che ci fornisce informazioni utili per il debug del programma.

Errori nei programmi eseguiti

Vediamo ora cosa accade se si prova ad invocare `ls` con un parametro errato. Questa situazione è nuova: il programma esiste e dovrebbe essere quindi eseguito dalla `exec` ma poi gli si passa un parametro che lo manda in errore.

```
if (execlp("ls","ls","-z",NULL) == -1) {
    perror("errore durante la exec");
    // eventualmente si esce: exit(EXIT_FAILURE);
    }
```

dà il seguente output

```
provo a eseguire ls
ls: invalid option -- z
Try `ls --help' for more information.
```

Nota: la `exec` **non** fallisce (non ritorna e non esegue nessuna istruzione del programma che l'ha invocata). L'errore a terminale è prodotto dalla `ls`. Quindi non è la `exec` a fallire ma il programma `ls` il quale ha già sovrascritto codice e dati del programma originale. Non c'è modo di gestire l'errore della `ls` dal programma chiamante.

Un esempio completo: simulare una shell

Vediamo come si può simulare il comportamento di una shell semplificando un po' la gestione dei parametri (non ci interessa fare il parse dell'input ma solo mostrare la generazione del nuovo processo e la sua esecuzione)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
main() {
    int esito;
    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);
            }
        // il processo genitore (shell) torna immediatamente a leggere un altro comando
    }
}
```

Proviamo a compilarlo e eseguirlo:

```
focardi@sally$ myshell
myshell# ls
myshell# exec1.c   exec2.c~  fork-fork-fork.c   shell.c~
exec1.c~  exec3.c   fork-fork-fork.c~
```

Cosa si nota di anomalo?

Non attende la terminazione del processo figlio: è come avere un `&` implicito. Lo si nota anche dal prompt `myshell#` immediatamente prima della lista dei file generata da `ls`: la shell chiede subito il comando successivo senza aspettare il risultato di quello precedente. Per poter attendere e gestire la terminazione di un processo figlio dobbiamo vedere un po' più in dettaglio cosa accade quando un processo termina.

Terminazione di un processo

La terminazione di un processo rilascia le risorse allocate dal SO al momento della creazione (ad esempio la memoria e i file aperti) e "segnala" la terminazione al genitore: alcune informazioni di stato vengono messe a disposizione al processo genitore e devono rimanere memorizzate finché non vengono processate. Parte della informazioni contenute nella PCB vengono quindi mantenute dopo la terminazione, finché il processo genitore non ha eventualmente letto tali informazioni.

Il sistema mantiene almeno:

1. il PID,
2. lo stato di terminazione;
3. il tempo di CPU utilizzato dal processo.

Esistono due chiamate a sistema:

- `exit`: termina il processo (già usata negli esempi per i casi di errore);
- `wait`: attende una `exit` di un figlio (se uno dei figli è uno *zombie* ritorna subito senza bloccarsi).

NOTA. Un processo può anche terminare in modo anomalo a causa di un errore o perchè terminato dal SO o da altri processi.

Sintassi

- `exit(int stato)`: termina il processo ritornando lo `stato` al genitore; Si usano le costanti `EXIT_FAILURE` e `EXIT_SUCCESS` che normalmente sono uguali ad 1 e 0 rispettivamente;
- `pid = wait(int &stato)`: ritorna il `pid` e lo `stato` del figlio che ha terminato. Si invoca `wait(NULL)` se non interessa lo stato. Se non ci sono figli ritorna -1.

Valore di ritorno della wait

Lo stato titornato da `wait` va gestito con opportune macro:

- `WIFEXITED(status)==true` se il figlio è uscito normalmente con una `exit`.
  `WEXITSTATUS(status)` ritorna gli 8 bit di stato passati dalla `exit`.

Valore di ritorno della wait

Lo stato titornato da `wait` va gestito con opportune macro:

- `WIFEXITED(status)==true` se il figlio è uscito normalmente con una `exit`.
  `WEXITSTATUS(status)` ritorna gli 8 bit di stato passati dalla `exit`.

  Esempio di codice:

```
if (WIFEXITED(status))
    printf("OK: status = %d\n",WEXITSTATUS(status));
```

- `WIFSIGNALED(status)==true` se il figlio è stato terminato in maniera anomala.
  `WTERMSIG(status)` ritorna il "segnale" che ha causato la terminazione.

  Esempio di codice:

```
if (WIFSIGNALED(status))
    printf("ANOMALO: status = %d\n",WTERMSIG(status));
```

- macro analoghe per stop/resume, utili per il tracing dei processi (`WIFSTOPPED`, `WSTOPSIG`, `WIFCONTINUED`).

Variante per attendere un processo particolare

Se si vuole attendere un processo particolare (o processi appartenenti a un gruppo particolare), si può utilizzare la chiamata a sistema `pid = waitpid(pid2,&stato,options)`

- attende il processo `pid2` (valori di `pid2` minori o uguali a zero permettono di attendere gruppi di processi, vedere il manuale per maggiori dettagli);
- se `pid2 == -1` attende un qualsiasi figlio: diventa uguale alla `wait`;

Esempio completo con exit e wait

Vediamo le chiamate a sistema `exit` e `wait` in azione nel seguente codice. Vengono creati 2 figli: il primo termina normalmente restituendo un valore al genitore, il secondo dereferenzia l'indirizzo 0 generando un segmentation fault. Il processo genitore attende la terminazione dei figli e stampa le relative informazioni.

```
#include<wait.h>
#include<stdio.h>
#include<stdlib.h>
main() {
        int pid,status;

        printf("pid = %d e pid della shell = %d\n",getpid(), getppid());

        if ((pid = fork())<0) {
                perror("errore fork"); exit(1); }

        /* figlio 1: esce normalmente inviando al genitore lo stato "42" */
        else if (pid == 0) {
                printf("Sono il figlio1! pid=%d ppid=%d\n",getpid(), getppid());
                sleep(3);
                exit(42);}

        if ((pid = fork())<0) {
                perror("errore fork"); exit(1); }

        /* figlio 2: segfault, cerca di accedere alla locazione 0 */
        else if (pid == 0) {
                int *tmp=0;
                int a;
                printf("Sono il figlio2! pid=%d ppid=%d\n",getpid(), getppid());
                sleep(5);
                a = *tmp; // segfault}

        /* solo il genitore continua e attende tutti i figli ... */
        while((pid=wait(&status)) >= 0) {
        printf("ricevuta terminazione di pid=%d\n",pid);
        if (WIFEXITED(status))
             printf("OK: status = %d\n",WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
             printf("ANOMALO: status = %d\n",WTERMSIG(status));
        }
}
```

dà il seguente output:

```
pid = 10080 e pid della shell = 5884
Sono il figlio1! pid=10081 ppid=10080
Sono il figlio2! pid=10082 ppid=10080
ricevuta terminazione di pid=10081
OK: status = 42
ricevuta terminazione di pid=10082
ANOMALO: status = 11
```

Il codice 11 corrisponde al 'segnale' di violazione di segmento (vedremo più in dettaglio i segnali nella prossima lezione).

Esercizio. Aggiungere le opportune `wait`, con relativa gestione dello stato, al codice della shell visto precedentemente. Provare anche ad eseguire, tramite tale shell, programmi che effettuano errori run-time (come divisioni per zero).

Esercizio. Una tecnica per creare *demoni* (programmi in esecuzione che non sono sotto il controllo degli utenti, come i servizi di sistema) è quella di eseguire una doppia `fork` e far terminare il primo figlio: in questo modo il processo "nipote" viene adottato da `init` e si distacca dal processo "nonno" definitivamente.
Scrivere il codice che realizzi questa tecnica facendo attenzione che

1. la seconda `fork` vegna effettuata solo dal primo figlio e non dal genitore, altrimenti si generano due nuovi processi;
2. la terminazione del primo figlio venga correttamente processata dal genitore tramite una opportuna `wait` (meglio ancora una `waitpid`), onde evitare processi *zombie*.

Soluzione esercizio sulla fork

L'output è una permutazione qualunque del seguente:

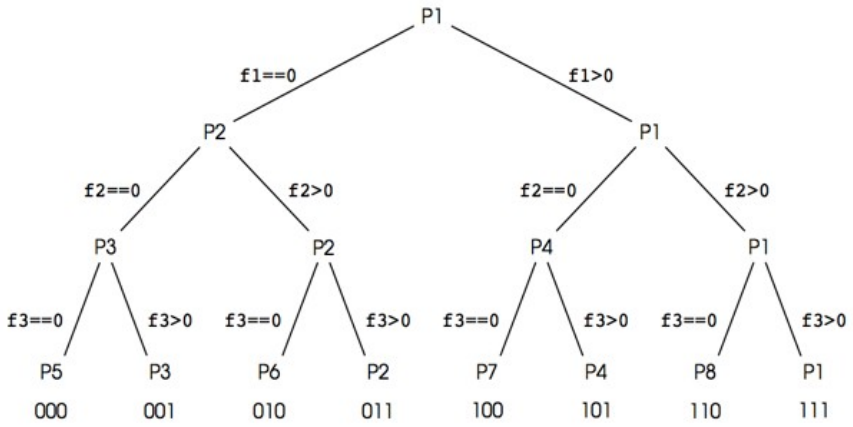
```
000 001 100 101 010 011 110 111
```

cioè tutti i numeri binari di 3 cifre in qualche ordine (dipende dallo scheduling).

Nota: esecuzioni diverse possono dare ordinamenti diversi: provare a eseguire più volte il programma e osservare l'output.

Nota: esecuzioni diverse possono dare ordinamenti diversi: provare a eseguire più volte il programma e osservare l'output.

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



Segnali

I segnali costituiscono una forma (molto 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, riferimento "sbagliato" in memoria, ecc…
- segnali esplicitamente inviati da un processo all'altro.
- eventi asincroni che vengono notificati ai processi: esempio `SIGALARM`.

Cosa possiamo fare quando arriva un segnale?

- Ignorarlo
- Gestirlo
- Lasciare il compito al gestore di sistema

Vediamo alcuni segnali POSIX (Portable Operating System Interface) supportati in Linux. La lista qui sotto è contenuta in `'man 7 signal'`

Vediamo alcuni segnali POSIX (Portable Operating System Interface) supportati in Linux. La lista qui sotto è contenuta in `'man 7 signal'`

```
First the signals described in the original POSIX.1-1990 standard.

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
SIGTSTP   18,20,24    Stop    Stop typed at tty
SIGTTIN   21,21,26    Stop    tty input for background process
SIGTTOU   22,22,27    Stop    tty output for background process

The  signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
```

Azioni possibili:

```
The  entries  in  the  "Action"  column of the tables below specify the
default disposition for each signal, as follows:

Term   Default action is to terminate the process.

Ign    Default action is to ignore the signal.

Core   Default action is to terminate the process and dump core (see core(5)).

Stop   Default action is to stop the process.

Cont   Default action is to continue the process if it is currently stopped.
```

Un esempio semplice: alarm

`alarm` manda un `SIGALRM` dopo n secondi. Il default handler scrive "alarm clock" e poi termina il programma (serve per dare un timeout).

Vediamo un semplice esempio di programma che setta l'allarme dopo 3 secondi e poi va in loop infinito:

```
main()
{
        alarm(3);
        while(1){}
}
```

Il programma "punta una sveglia" dopo 3 secondi e attende in un ciclo infinito. Allo scadere dei 3 secondi viene inviato un segnale `SIGALRM` al processo. Il comportamento di default è quello di terminare il processo: se proviamo ad eseguire il programma osserviamo, infatti, che dopo 3 secondi il processo termina (viene anche scritto a terminale il motivo della terminazione).

```
$ a.out
<... dopo 3 secondi....>
Alarm clock
$
```

Impostare il gestore dei segnali tramite 'signal'

Tramite la system call `signal` è possibile cambiare il gestore dei segnali. La system call prende come parametri un segnale e una funzione che da quel momento diventerà il nuovo gestore del segnale. Vediamo un semplice esempio:

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

void alarmHandler()
{
        printf("questo me lo gestisco io!\n");
        alarm(3); // ri-setta il timer a 3 secondi
}

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

Dopo tre secondi arriva il segnale `SIGALRM`. Viene invocato `alarmHandler` che stampa a video una frase e reimposta l'allarme dopo 3 secondi. Il controllo ritorna al punto in cui il programma è stato interrotto (nel ciclo while quindi) e il programma attende altri 3 secondi che arrivi il successivo segnale:

```
questo me lo gestisco io!       <=== dopo 3 secondi
questo me lo gestisco io!       <=== dopo 3 secondi
questo me lo gestisco io!       <=== dopo 3 secondi
........
```

Parametri particolari e valore di ritorno

È possibile passare alla `signal` le costanti `SIG_IGN` o `SIG_DFL` al posto della funzione handler per indicare, rispettivamente:

- che il segnale va ignorato
- che l'handler è quello di default di sistema

Il valore di ritorno di `signal` è

- `SIG_ERR` in caso di errore
- l'handler precedente, in caso di successo

NOTA: `sigaction` rimpiazza `signal` con un'implementazione più stabile nelle varie versioni UNIX. Viene raccomandata se si vuole portabilità. Ad esempio, la `signal` originale UNIX, e la sua versione System V, fa il reset del gestore a `SIG_DFL` ogni volta che viene ricevuto il segnale. In Linux, si ottiene questo comportamento particolare compilando con l'opzione `--ansi`. Per l'uso di `sigaction` si faccia riferimento al manuale.

Esempio: Proteggersi da ctrl-c

Vediamo ora un altro esempio di gestione dei segnali. Se modifichiamo il gestore del segnale `SIGINT` possiamo evitare che un programma venga interrotto tramite ctrl-c da terminale.

```
#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);
}
```

Notare l'uso del valore di ritorno della signal 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`.

Se eseguiamo il programma possiamo osservare che per 3 secondi ctrl-c non ha alcun effetto. Appena viene reimpostato il vecchio gestore, invece, ctrl-c interrompe il programma.

Se eseguiamo il programma possiamo osservare che per 3 secondi ctrl-c non ha alcun effetto. Appena viene reimpostato il vecchio gestore, invece, ctrl-c interrompe il programma.

```
> a.out
Sono protetto!
<ctrl-c>
<ctrl-c>
<ctrl-c>                <==== nessun effetto
Non sono più protetto!
<ctrl-c>                <==== esce!
>
```

Sospensione e ripristino di processi tramite kill

La chiamata a sistema `kill` manda un segnale a un processo.

In questo esempio mostriamo come la chiamata a sistema `kill` possa essere utilizzata per sospendere, ripristinare e terminare un processo.

In questo esempio mostriamo come la chiamata a sistema `kill` possa essere utilizzata per sospendere, ripristinare e terminare un processo.

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <signal.h>
main(){
        pid_t pid1,pid2;

        if ( (pid1 = fork()) < 0 ) {
                perror("errore fork"); exit(EXIT_FAILURE);
        } else if (pid1 == 0)
                while(1) { // primo figlio
                        printf("%d è vivo !\n",getpid());
                        sleep(1);
                        }

        if ( (pid2 = fork()) < 0 ) {
                perror("errore fork"); exit(EXIT_FAILURE);
        } else if (pid2 == 0)
                while(1) { // secondo figlio
                        printf("%d è vivo !\n",getpid());
                        sleep(1);
                        }
        // processo padre
        sleep(2);
        kill(pid1,SIGSTOP); // sospende il primo figlio
        sleep(5);
        kill(pid1,SIGCONT); // risveglia il primo figlio
        sleep(2);
        kill(pid1,SIGINT); // termina il primo figlio
        kill(pid2,SIGINT); // termina il secondo figlio
}
```

Quando eseguiamo il programma notiamo che il primo figlio viene sospeso per 5 secondi e che alla fine entrambi i processi figli sono terminati:

```
> a.out
6720 è vivo !
6721 è vivo !
6720 è vivo !
6721 è vivo !
6720 è vivo !
6721 è vivo !    <==== sospende 6720
6721 è vivo !
6721 è vivo !
6721 è vivo !
6721 è vivo !
6720 è vivo !    <==== risveglia 6720
6721 è vivo !
6720 è vivo !
6721 è vivo !
>
```

Mascherare i segnali

A volte risulta utile bloccare temporaneamente la ricezione dei segnali per poi riattivarli. Tali segnali non sono ignorati ma solamente 'posticipati'.

NOTA POSIX non specifica se più occorrenze dello stesso segnale debbano essere memorizzate (accodate) oppure no. Tipicamente se più segnali uguali vengono generati, solamente uno verrà "recapitato" quando il blocco viene tolto.

La chiamata a sistema `sigprocmask(int action, sigset_t *newmask, sigset_t *oldmask)` compie azioni differenti a seconda del valore del primo parametro `action`:

- `SIG_BLOCK`: l'insieme dei segnali `newmask` viene unito all'insieme dei segnali attualmente bloccati, che sono restituiti in `oldmask`;
- `SIG_UNBLOCK`: l'insieme dei segnali `newmask` viene sottratto dai segnali attualmente bloccati, sempre restituiti in `oldmask`;
- `SIG_SETMASK`: l'insieme dei segnali `newmask` sostituisce quello dei segnali attualmente bloccati (`oldmask`).

Per gestire gli insiemi di segnali (di tipo `sigset_t`) si utilizzano:

La chiamata a sistema `sigprocmask(int action, sigset_t *newmask, sigset_t *oldmask)` compie azioni differenti a seconda del valore del primo parametro `action`:

- `SIG_BLOCK`: l'insieme dei segnali `newmask` viene unito all'insieme dei segnali attualmente bloccati, che sono restituiti in `oldmask`;
- `SIG_UNBLOCK`: l'insieme dei segnali `newmask` viene sottratto dai segnali attualmente bloccati, sempre restituiti in `oldmask`;
- `SIG_SETMASK`: l'insieme dei segnali `newmask` sostituisce quello dei segnali attualmente bloccati (`oldmask`).

Per gestire gli insiemi di segnali (di tipo `sigset_t`) si utilizzano:

- `sigemptyset(sigset_t *set)` che inizializza l'insieme `set` all'insieme vuoto
- `sigaddset(sigset_t *set, int signum)` che aggiunge il segnale `signum` all'insieme set

L'esempio mostra come bloccare `SIGINT` e poi ripristinarlo.

```c
#include<signal.h>
#include<stdio.h>
#include<stdlib.h>
main() {
        sigset_t newmask,oldmask;

        sigemptyset(&newmask);          // insieme vuoto
        sigaddset(&newmask, SIGINT);    // aggiunge SIGINT alla "maschera"
        // setta la nuova maschera e memorizza la vecchia
        if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) {
                perror("errore settaggio maschera"); exit(1); }

        printf("Sono protetto!\n");
        sleep(3);

        // reimposta la vecchia maschera
        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) {
                perror("errore settaggio maschera"); exit(1); }

        printf("Non sono piu' protetto!\n");
        sleep(3);
}
```

Se digitiamo ctrl-c mentre il segnale è mascherato esso viene sospeso. Appena la maschera viene rimossa, il segnale è ricevuto dal processo che viene immediatamente interrotto. Non stiamo quindi modificando il gestore del segnale ma solo sospendendone la ricezione per un periodo di tempo.

```
> a.out
Sono protetto!
<ctrl-c>
<ctrl-c>
<ctrl-c>                <==== per 3 secondi nessun effetto
                        <==== esce appena la maschera viene tolta
>                               (senza dare ulteriori ctrl-c)!
```

Tramite `sigpending` è possibile ottenere l'insieme dei segnali "pendenti" (vedere il manuale per maggiori informazioni)

NOTA: `sigprocmask` non aderisce allo standard ANSI (ISO C99) ma solamente allo standard POSIX. La gestione dei segnali da parte di ANSI-C è molto povera e si riduce solamente a `signal`, `raise(sig)`, che equivale a `kill(getpid(),sig)`, e `abort()`, cioè "abnormal termination". ANSI-C non prevede multi-processing e quindi non prevede l'invio di segnali ad altri processi (sezione 7.14.2.1).

Attendere un segnale tramite 'pause'

Negli esempi abbiamo sempre usato `while(1){}` per attendere un segnale (*busy-waiting*). La system call `pause()` attende un segnale senza consumare tempo di CPU. Vediamo un esempio:

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

void alarmHandler()
{
        printf("questo me lo gestisco io!\n");
}

main()
{
        signal(SIGALRM, alarmHandler);
        alarm(3);
        pause();
        printf("termino!\n");
}
```

Interferenze e funzioni 'safe' POSIX

L'uso della `printf` nell'handler è rischioso perchè usa *dati globali*: Se anche il programma interrotto stava facendo I/O i due potrebbero interferire! printf non è "safe".

Facendo `man 7 signal` (in sistemi Linux recenti), troviamo la lista di funzioni safe che sicuramente **non** creano problemi di interferenza:

```
       _Exit(),  _exit(),  abort(),  accept(),  access(),  aio_error(),  aio-
       _return(),  aio_suspend(),  alarm(),  bind(),  cfgetispeed(),  cfget-
       ospeed(),  cfsetispeed(),  cfsetospeed(),  chdir(),  chmod(),  chown(),
       clock_gettime(), close(), connect(), creat(), dup(), dup2(), execle(),
       execve(),  fchmod(),  fchown(),  fcntl(),  fdatasync(),  fork(), fpath-
       conf(), fstat(), fsync(), ftruncate(), getegid(), geteuid(), getgid(),
       getgroups(),  getpeername(),  getpgrp(),  getpid(),  getppid(),  get-
       sockname(), getsockopt(), getuid(), kill(), link(), listen(), lseek(),
       lstat(),  mkdir(),  mkfifo(),  open(),  pathconf(),  pause(),  pipe(),
       poll(), posix_trace_event(), pselect(), raise(), read(), readlink(),
       recv(), recvfrom(), recvmsg(), rename(), rmdir(), select(), sem_post(),
       send(), sendmsg(), sendto(), setgid(), setpgid(), setsid(), setsock-
       opt(),  setuid(),  shutdown(),  sigaction(),  sigaddset(), sigdelset(),
       sigemptyset(), sigfillset(), sigismember(), signal(), sigpause(), sig-
       pending(),  sigprocmask(), sigqueue(), sigset(), sigsuspend(), sleep(),
       socket(),  socketpair(),  stat(),  symlink(),  sysconf(),  tcdrain(),
       tcflow(),  tcflush(),  tcgetattr(),  tcgetpgrp(), tcsendbreak(), tcset-
       attr(),  tcsetpgrp(),  time(),  timer_getoverrun(),  timer_gettime(),
       timer_settime(),  times(), umask(), uname(), unlink(), utime(), wait(),
       waitpid(), write().
```

Il seguente esempio cerca di far interferire le `printf` aggiungendo nel ciclo while la stampa di una stringa. Se il programma viene interrotto proprio durante la stampa, la printf del gestore potrebbe interferire con quella del programma.

Il seguente esempio cerca di far interferire le `printf` aggiungendo nel ciclo while la stampa di una stringa. Se il programma viene interrotto proprio durante la stampa, la printf del gestore potrebbe interferire con quella del programma.

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

void alarmHandler();    // gestore
static int i=0;         // contatore globale `volatile'

main() {
        signal(SIGALRM, alarmHandler);
        alarm(1);
        while(1){
                printf("prova\n");
        }
}
void alarmHandler()
{
        printf("questo me lo gestisco io %d!\n",i++);
        alarm(1);       // ri-setta il timer a 1 secondo
}
```

Tipicamente le stampe sono troncate o mischiate. In alcuni casi si possono addirittura perdere alcune `printf` (e/o alcuni a-capo) perché eseguite a metà di un'altra `printf`.

NOTA è necessario eseguire il comando con in coda " `| grep io`" per evitare di osservare tutte le stampe "prova". Il comando '`grep`' stampa solo le linee contenenti la stringa "io" (che compare in "questo me lo gestisco io"). la *pipe* '|' fa si che l'output del comando venga reindirizzato al comando successivo come input (è un modo per creare 'al volo' un canale di comunicazione *message passing* tra due processi, che approfondiremo la prossima volta).

NOTA è necessario eseguire il comando con in coda " | `grep io`" per evitare di osservare tutte le stampe "prova". Il comando '`grep`' stampa solo le linee contenenti la stringa "io" (che compare in "questo me lo gestisco io"). la *pipe* '|' fa si che l'output del comando venga reindirizzato al comando successivo come input (è un modo per creare 'al volo' un canale di comunicazione *message passing* tra due processi, che approfondiremo la prossima volta).

```
> a.out | grep io
questo me lo gestisco io 0!
questo me lo gestisco io 2!
questo me lo gestisco io 4!
questo me lo gestisco io 6!
questo me lo gestisco io 7!
questo me lo gestisco io 8!
questo me lo gestisco io 12!
questo me lo gestisco io 14!
questo me lo gestisco io 15!
questo me lo gestisco io 16!
questo me lo gestisco io 17!
provaquesto me lo gestisco io 18!
provaquesto me lo gestisco io 19!
```

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. Vediamo alcuni esempi:

- **Starvation**: un processo è bloccato indefinitamente a causa di altri processi che monopolizzano una o più risorse. Abbiamo visto un esempio estremo di programma "fork-bomb" che se non limitato opportunamente può arrivare a sottrarre tutte le risorse agli altri processi:

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

- **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. Ad esempio:

- Le stampe su una stampante devono essere accodate e non "mischiate" in modo incontrollato;
- L'accesso al disco deve garantire che diversi processi non scrivano inavvertitamente sulla stesso blocco;
- L'accesso alla CPU deve essere regolato da un timer che la assegna in modo sufficientemente equo ai diversi processi in esecuzione.

A tale scopo, il sistema operativo **virtualizza** l'hardware in modo da dare l'impressione che ogni processo abbia una istanza dedicata della risorsa. Quando scriviamo sul disco, a parte la velocità di accesso, non ci accorgiamo se altri processi stanno leggendo o scrivendo su altri file. Allo stesso modo, non ci accorgiamo di altri processi in esecuzione sulla CPU, fintantoché questo non rallenta visibilmente l'esecuzione dei nostri programmi.

Cooperazione

Ci sono molti casi in cui l'interazione è però voluta. Vediamo alcuni esempi:

- **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.
  I comandi Unix sono un altro tipico esempio di comportamento modulare:

```
$ ls -al | grep pippo
-rw-rw-r--   1 focardi focardi       0 Jul 12 12:48 pippo.html
-rw-r--r--   1 focardi focardi    2014 Feb 13  2012 pippo.txt
$
```

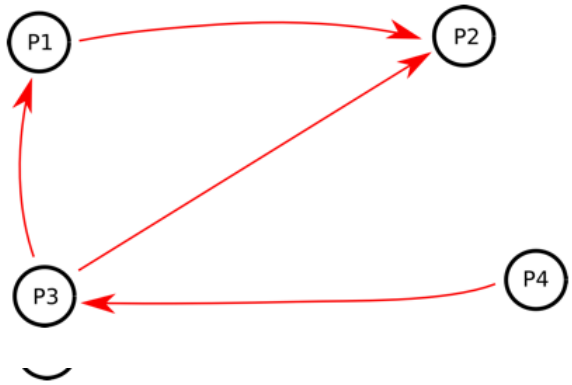
Il comando `ls -al` mostra il contenuto del folder. Il suo output viene dato in input a un secondo comando `grep pippo` che stampa solo le righe contenenti la parola `pippo`. Il simbolo "|" (pipe) indica appunto che l'output del primo comando deve essere dato in input al secondo. In questo modo otteniamo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

- **Convenienza**: può risultare comodo eseguire un'attività mentre si continua con un'altra. Vogliamo ad esempio lanciare una stampa e continuare a lavorare sul documento mentre il file viene stampato;
- **Replicazione**: quando è necessaria l'esecuzione simultanea di diverse istanze di un'attività diventa comodo replicarla su più processi o thread. Un esempio tipico è un server che si replica per servire diversi utenti contemporaneamente. Ogni processo servente dovrà tipicamente coordinarsi con gli altri nell'accesso alle risorse comuni.

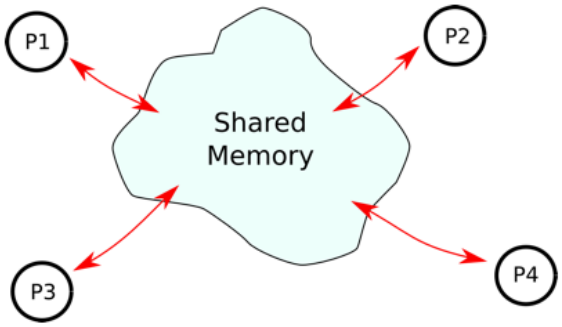
Modelli di comunicazione

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

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



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



In questa prima parte del corso ci interessiamo allo scambio di messaggi in quanto costituisce il modello di riferimento per la comunicazione tra processi. Nella seconda parte del corso discuteremo approfonditamente il modello a memoria condivisa che è, invece, il modello di riferimento per la comunicazione tra thread. I thread, infatti, condividono tutte le risorse del processo, tra cui anche la memoria.

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 nelle primitive:

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

È come se esistesse **un canale riservato** per ogni coppia di processi. Basta conoscere la reciproca identità. Esiste anche una variante asimmetrica in cui si riceve da qualsiasi utente

- receive(**&Q**,&m), riceve un messaggio m da un qualsiasi utente. Messaggio e mittente vengono salvati in m e **Q**.

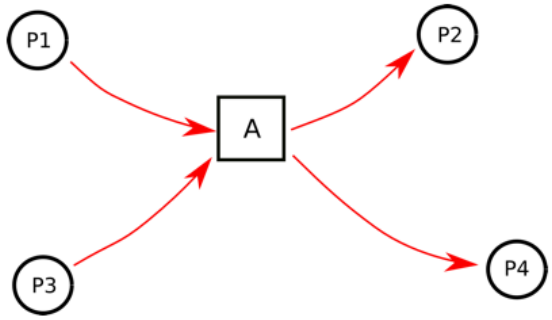
Vantaggi: la nominazione diretta è molto semplice e permette di comunicare in modo diretto tra coppie di processi;
Svantaggi: È però necessario un "accordo" sui nomi dei processi. Il PID infatti viene dato dinamicamente dal sistema e non possiamo prevederne il valore a priori. Nella pratica è difficile da implementare a meno che i processi non siano in relazione stretta di parentela genitore-figlio. In tale caso, infatti, il genitore conosce il PID dei vari figli e potrebbe comunicare direttamente con loro utilizzando tale identificativo. Vedremo, però, che nel caso di processi parenti non è nemmeno necessario utilizzare questo tipo di nominazione.

Nominazione indiretta

Per ovviare ai difetti della nominazione diretta si utilizza, in pratica, una nominazione indiretta tramite **porte** (o **mailbox**). 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.



Competizione: cosa accade se due processi cercano di inviare o ricevere messaggi dalla stessa porta? In questo caso sarà il sistema operativo a gestire la competizione in modo opportuno, se possibile. Spesso si associa una porta a un processo (il creante e possessore della porta) che è l'unico a leggere da tale porta. Programmando in questo modo si evitano problemi legati alla competizione in ricezione. L'invio è meno problematico in quanto i messaggi possono essere accodati sulla porta.

Nel prossimo laboratorio vedremo come le pipe di Unix implementino un meccanismo di scambio di messaggi a nominazione indiretta. Vedremo inoltre che 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 sincrona e asincrona

Invio e ricezione possono essere sincroni o asincroni.

- *send sincrona*: il messaggio viene inviato solo quando la corrispondente receive viene eseguita. La send è quindi bloccante fintantoché il messaggio non viene ricevuto. Un esempio può essere una telefonata: si inizia a parlare solo quando dall'altro lato c'è un interlocutore. Lo stesso avviene in una chat;
- *send asincrona*: il messaggio viene inviato indipendentemente dalla receive. Per realizzare una send asincrona è necessaria una **coda** (buffer) in cui depositare temporaneamente il messaggio. Un esempio può essere la posta che viene depositata nella buchetta e, analogamente, la email che viene inviata indipendentemente dalla presenza online del ricevente;
- *receive sincrona*: il messaggio viene ricevuto solo se presente. La receive è bloccante fintantoché non c'è un messaggio da leggere. Un server web in attesa di connessioni è un esempio di receive sincrona.

- *receive asincrona*: la receive ritorna un messaggo se presente o NULL se non ci sono messaggi da ricevere. Un client di email che periodicamente verifica la presenza di nuovi messaggi è un esempio di receive asincrona: se non ci sono nuovi messaggi il client prosegue con quelli presenti, altrimenti li aggiorna.

Produttore-consumatore

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:

```
$ ls -al | grep pippo
-rw-rw-r--   1 focardi focardi       0 Jul 12 12:48 pippo.html
-rw-r--r--   1 focardi focardi    2014 Feb 13  2012 pippo.txt
$
```

Il processo `ls -al` produce un output che viene dato in input a `grep pippo` che lo "consuma" generando un ulteriore output.

Questo esempio mostra anche una realizzazione della comunicazione a nominazione indiretta con send asincrona e receive sincrona. Vediamo in dettaglio:

1. la shell crea **due processi** figlio "`ls -al`" e "`grep pippo`";
2. la shell crea inoltre una **pipe** indicata dal simbolo "|". La creazione della pipe avviene prima delle fork. La pipe quindi è condivisa dai processi figli;
3. tale pipe diventa l'equivalente di una porta: i due processi figlio possono nominarla e utilizzarla per comunicare: "`ls -al`" manda tutto il suo output sulla pipe mentre "`grep pippo`" legge il proprio input dalla pipe;
4. la pipe ha un buffer che consente al primo processo di inviare in modo asincrono i dati.
   Per illustrare questo comportamento aggiungiamo una sleep prima di "grep" e mettiamo una stampa sullo standard error dopo "ls" (per deviare lo standard output sullo standard error è sufficiente usare la sintassi 1>&2 che ridirezione 1, stdout, su 2, stderr).

   ```
   $ (ls -al; echo "DONE ls" 1>&2) | (sleep 10;grep pippo)
   DONE ls
   -rw-rw-r--   1 focardi focardi       0 Jul 12 12:48 pippo.html
   -rw-r--r--   1 focardi focardi    2014 Feb 13  2012 pippo.txt
   $
   ```

   Notiamo che "DONE ls" viene stampato immediatamente mentre la stampa delle linee contenenti pippo avviene dopi 10 secondi. Significa quindi che "ls -al" ha terminato l'esecuzione immediatamente stampando "DONE ls" e il suo output (il contenuto del folder) è stato bufferizzato dalla pipe. Quando la sleep termina tale output viene letto dalla pipe e processato da "grep";

5. la ricezione dalla pipe è, di default, sincrona: il secondo processo attende fintantoché non ci sono dati da leggere.

   ```
   $ (sleep 2;ls -al) | grep pippo
   -rw-rw-r--   1 focardi focardi       0 Jul 12 12:48 pippo.html
   -rw-r--r--   1 focardi focardi    2014 Feb 13  2012 pippo.txt
   $
   ```

   In questo caso la sleep è prima di "ls" e quindi l'esecuzione è ritardata di 2 secondi. Il comando "grep" è in attesa sulla pipe e aspetta anch'esso l'esecuzione della sleep. La ricezione è infatti sincrona.

Pipe

Le *pipe* sono la forma più "antica" di comunicazione tra processi UNIX. Una pipe, che letteralmente significa *tubo*, costituisce un vero e proprio 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) con send asincrona e receive sincrona.

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). Prima di proseguire con questa esercitazione vi consiglio di provare i semplici esempi illustrati in aula la lezione scorsa.

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

- `filedes[0]` per la lettura;
- `filedes[1]` per la scrittura;

Notiamo quindi che le pipe 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 il seguente esempio:

```
#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 */
        }
}
```

Descrizione: Viene creata una pipe e, subito dopo, un nuovo processo. Le pipe vengono ereditate dai figli e quindi entrambi i processi, dopo la fork, condividono la pipe. Il processo figlio invia una stringa (incluso il terminatore 0×00) e il processo padre la legge. Notare che il processo che scrive chiude subito la pipe in lettura mentre quello che legge la chiude in scrittura. Questo è molto importante: ogni processo tiene aperte solo le risorse che intende utilizzare.

Se eseguiamo il programma otteniamo il seguente output:

```
$ ./prova_pipe
ho letto dalla pipe 24 bytes: 'prova a inviare questo!'
$
```

Vediamo che il processo padre riceve correttamente il messaggio. Come nella lettura da file, la read restituisce il numero di byte letti.

Invio e ricezione su una pipe "chiusa"

Ci sono alcune situazioni, tipiche delle pipe, che analizziamo in dettaglio:

- *Cosa accade se si fa una read da una pipe che è vuota ed è stata chiusa in scrittura (non ci sono più dati nel buffer e non ci sono più scrittori)?*
  La read ritorna 0, corrispondente a un end-of-file.

- *Cosa accade se si fa una write su una pipe che è stata chiusa in lettura (non ci sono più lettori)?*
  Viene generato il segnale `SIGPIPE` che di default termina il processo. Se si ignora o si gestisce il segnale la write ritorna un errore e `errno` è settata a `EPIPE`

NOTA: Qui si capisce l'importanza di **chiudere subito** una risorsa che non verrà utilizzata. Se il processo che legge solamente non chiude in scrittura la pipe, tale processo non riceverà l'end-of-file nel caso l'altro processo chiuda la pipe in scrittura. Infatti, esiste ancora un processo scrittore attivo e il sistema tiene aperta la pipe. Lo stesso accade con `SIGPIPE` se il processo che scrive solamente non chiude la pipe il lettura.

ESERCIZIO: modificare il programma precedente in modo da osservare i due eventi discussi qui sopra (assenza di scrittori, assenza di lettori)

Esempio: La pipe della shell

La seguente linea di comando prende l'output di `prog1` e lo manda in input a `prog2`.

```
$ prog1 | prog2
```

Per capire come la shell implementa questo comportamento proviamo a simularlo in C:

```c
#include <stdio.h>
#include <unistd.h>
main(int argc, char * argv[])
{
        int fd[2], bytesread;

        pipe(fd);
        if (fork() == 0)        {
                close(fd[0]);                                   /* chiude in lettura */
                dup2 (fd[1],1);                                 /* fa si che 1 (stdout) sia una copia di fd[1] */
                                                                /* da qui in poi l'output va sulla pipe */
                close(fd[1]);                                   /* chiude il descrittore fd[1] */
                execlp(argv[1],argv[1],NULL);                   /* esegue il comando */
                perror("errore esecuzione primo comando");
        } else {
                close(fd[1]);                                   /* chiude in scrittura */
                dup2 (fd[0],0);                                 /* fa si che 0 (stdin) sia una copia di fd[0] */
                                                                /* da qui in poi l'input proviene dalla pipe */
                close(fd[0]);                                   /* chiude il descrittore fd[0] */
                execlp(argv[2],argv[2],NULL);                   /* esegue il comando */
                perror("errore esecuzione secondo comando");
        }
}
```

Descrizione: Viene creata una pipe e viene eseguita una fork. I due processi ereditano la pipe e chiudono i descrittori che non useranno. Successivamente "copiano" i descrittori di scrittura e lettura della pipe, rispettivamente, sullo standard output e input (vedi la nota sotto). Poi chiudono definitivamente i descrittori della pipe e fanno una exec.

NOTA: `dup2(fd[1],1)` fa si che il descrittore dello stdout (1) da questo momento in poi punti a ciò che è puntato da `fd[1]` (diventano intercambiabili). Tutto ciò che verrà mandato sullo standard output andrà a finire sulla pipe. Se 1 puntasse ad un normale file il file verrebbe chiuso, se non riferito da altri descrittori. Notare che dopo la dup2 possiamo chiudere il descrittore della pipe, in quanto non verrà più utilizzato.

Facciamo un test per vedere che il programma si comporta come ci attendiamo (il comando 'wc' word-count, conta il numero di newline, parole e byte date in input).

```
$ who
focardi  pts/0        2012-11-07 14:13 (sally.dsi.unive.it)
focardi  pts/1        2012-11-13 23:47 (sally.dsi.unive.it)
$ who | wc
      2      10     120
$ ./simula_pipe who wc
      2      10     120
$
```

Il programma simula quindi l'esecuzione con la pipe della shell. In realtà la shell esegue due fork, una per ogni processo eseguito e attende la terminazione. Per semplificare abbiamo simulato la pipe con una sola fork.

Atomicità

Letture e scritture sulle pipe sono atomiche se inferiori alla dimensione `PIPE_BUF` (`limits.h`), usualmente 4096 bytes. Sopra tale dimensione l'atomicità non è garantita. È importante ricordarsi questo aspetto perché sopra tale dimensione, se più processi scrivono contemporaneamente, non è detto che i byte in scrittura non si 'mischino' tra loro.

Pipe con nome

Le pipe senza nome non possono essere utilizzate da processi che non hanno un antenato in comune in quanto l'unico modo per leggere e scrivere è tramite i descrittori di lettura e scrittura. Per ovviare a questa limitazione esistono le 'pipe con nome'. Tali pipe possono essere create con il comando `mkfifo`.

```
$ mkfifo myPipe     <==== (oppure mknod myPipe p)
$ ls -al
totale 36
drwxrwxr-x    2 focardi  focardi      4096 mag 23 00:57 .
drwxrwxr-x    7 focardi  focardi      4096 mag 23 00:16 ..
...
prw-rw-r--    1 focardi  focardi         0 mag 23 00:57 myPipe
...
$
```

Da questo momento in poi la pipe esiste nel filesystem e qualsiasi processo con i diritti di accesso al file può utilizzarla.

Esempio: Un lettore e tanti scrittori

Consideriamo un processo *lettore* (destinatario) che accetta, su una pipe con nome, messaggi provenienti da più *scrittori* (mittenti). Gli scrittori mandano 3 messaggi e poi terminano. Quando tutti gli scrittori chiudono la pipe il lettore ottiene 0 come valore di ritorno dalla `read` ed esce. Lettori e scrittori sono processi distinti lanciati indipendentemente (non necessariamente parenti).

Processo lettore (vedere i commenti nel codice):

```c
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdlib.h>

#define PNAME "/tmp/aPipe"
main() {
        int fd;
        char str[100], leggi;

        mkfifo(PNAME,0666);     // crea la pipe con nome, se esiste gia' non fa nulla

        if ( (fd = open (PNAME,O_RDONLY)) < 0 ) { // apre la pipe in lettura
                perror("errore apertura pipe");
                exit(1);
        }

        while (read(fd,&leggi,1) ) {    // legge un carattere alla volta fino a EOF
                printf("%c",leggi);
                if (leggi == '\0') printf("\n"); // a capo dopo ogni stringa
        }

        close(fd);              // chiude il descrittore
        unlink(PNAME);          // rimuove la pipe
}
```

Processo scrittore (vedere i commenti nel codice):

```c
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>

#define PNAME "/tmp/aPipe"

main() {
        int fd,i;
        char message[100], leggi;

        mkfifo(PNAME,0666);     // crea la pipe con nome, se esiste gia' non fa nulla

        // crea la stringa da scrivere sulla pipe
        sprintf(message,"Saluti dal processo %d",getpid());

        if ( (fd = open(PNAME,O_WRONLY)) < 0) { // apre la pipe in scrittura
                perror("errore apertura pipe");
                exit(1);
        }

        for (i=1; i<=3; i++){           // scrive tre volte la stringa
                write (fd,message,strlen(message)+1);
                sleep(2);
                }

        close(fd);      // chiude il descrittore
}
```

Possiamo compilare i due processi e lanciarli indipendentemente. Lanciamo, ad esempio, il lettore e tre scrittori. Notare l'uso di & per lanciare i processi in 'background' (e quindi in parallelo).

```
> ./lettore & ./scrittore & ./scrittore & ./scrittore
[1] 46998
[2] 46999
[3] 47000
Saluti dal processo 47000
Saluti dal processo 46999
Saluti dal processo 47001
Saluti dal processo 47000
Saluti dal processo 47001
Saluti dal processo 46999
Saluti dal processo 47000
Saluti dal processo 46999
Saluti dal processo 47001
[1]   Done                    ./lettore
[2]-  Exit 255                ./scrittore
[3]+  Exit 255                ./scrittore
>
```

Notare che le scritte a video vengono effettuate dal lettore dopo aver ricevuto il messaggio dalla pipe. I processi scrittori non scrivono nulla a video. Provare anche a lanciare il lettore su un terminale e i tre scrittori su un terminale differente.

ESERCIZIO: Come abbiamo già discusso la write, sotto la dimensione `PIPE_BUF`, è atomica: Più processi possono scrivere messaggi sulla stessa pipe se tali messaggi sono più corti di `PIPE_BUF`: i messaggi saranno accodati uno dopo l'altro senza che i singoli caratteri si 'mescolino' tra loro. Il lettore invece legge un carattere alla volta. Provare a lanciare più lettori per osservare interferenze.

Le pipe con nome sono bidirezionali?

L'opzione `O_RDWR` nella open permette di aprire una pipe con nome in lettura e scrittura. Come si fa però a evitare che un processo legga ciò che lui stesso ha scritto? Questo fatto rende le pipe `O_RDWR` inutilizzabili in pratica. Tale modalità esiste solo per poter aprire una pipe in scrittura quando nessuno l'ha ancora aperta in lettura. Una open in scrittura di una pipe non ancora aperta in lettura è bloccante.

Altri esercizi sulle pipe

1. Le pipe gestiscono **stream di byte**: non c'è nessuna nozione di messaggio. Non è detto, quindi, che due write vengano lette tramite due read. È possibile sperimentare questo aspetto inviando due write distinte consecutive e osservando che vengono tipicamente lette da un'unica read (fare attenzione, se si inviano stringhe, a togliere il NULL dalla prima stringa in modo da 'concatenarle'. Altrimenti, le due stringhe verranno ricevute ma la `printf` visualizzerà solo i caratteri prima del primo NULL, ovvero la prima stringa).
2. Anche se non è ovvio da visualizzare, si può provare a superare `PIPE_BUF` (4096) per vedere che le write interferiscono una con l'altra. Si suggerisce di ridirezionare l'output del lettore su file (tramite '> nomefile') in modo da esaminare l'output con un editor alla ricerca di messaggi 'mescolati'

Esercitazione su pipe

Vediamo un esercizio sulle pipe (è una delle verifiche degli scorsi anni).

Crackme

Il programma crackme chiede un Personal Identification Number (PIN) di 5 cifre e ne verifica la correttezza. Si limita a stampare un messaggio ma si può pensare che solo nel caso il PIN sia corretto il programma dia accesso a risorse protette.

Una volta lanciato, crackme utilizza due pipe per interagire con altri processi. Su una pipe viene letto il PIN e sull'altra viene inviato il risultato della verifica in forma di stringa.

Il sorgente del programma (senza il PIN segreto) è il seguente:

```c
#include <fcntl.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>

#define PNAME1 "tmpPipeInput"
#define PNAME2 "tmpPipeOutput"
#define SUCCESSO "PIN corretto. Sei autenticato\n"
#define FALLIMENTO "PIN errato\n"
// il PIN è stato oscurato!

void chiuditutto() {
    unlink(PNAME1);        // rimuove la pipe
    unlink(PNAME2);        // rimuove la pipe
    exit(1);
}

// stampa l'errore e termina
void die(char *s) {
    perror(s);
    exit(EXIT_FAILURE);
}

main() {
    int fdI,fdO;
    int leggi;

    signal(SIGINT,chiuditutto);

    mkfifo(PNAME1,0666);    // crea la pipe, se esiste gia' non fa nulla
    mkfifo(PNAME2,0666);    // crea la pipe, se esiste gia' non fa nulla

    if ( (fdI = open (PNAME1,O_RDWR)) < 0 ) // apre la pipe per la lettura
        die("errore apertura pipe\n");

    if ( (fdO = open (PNAME2,O_RDWR)) < 0 ) // apre la pipe per la scrittura
        die("errore apertura pipe\n");

     while ( read(fdI, &leggi, sizeof(int) ) ) {
         // Controlla la correttezza del PIN
         if (leggi == PINsegreto)
             write(fdO, SUCCESSO, strlen(SUCCESSO)+1);
             // qui si ha accesso alle risorse ...
         else
             write(fdO, FALLIMENTO, strlen(FALLIMENTO)+
     }

}
```

Scaricare crackme qui.

Scrivere un programma crack.c che scopra il PIN segreto e lo stampi a video. Il programma deve interagire con crackme solo utilizzando le pipe (scoprire il PIN tramite debugging, anche se utile e divertente, non è considerata una soluzione, visto che l'esercizio è sull'uso delle pipe).

Notare che, per semplicità, gli interi vengono ricevuti direttamente nella loro rappresentazione binaria utilizzando la seguente istruzione:

```
read(fdI, &leggi, sizeof(int) )
```

Nel caso di interi a 32 bit, ad esempio, verranno letti direttamente i 4 byte che rappresentano il numero intero.

NOTA: È possibile interagire con le pipe di crackme da terminale ma si deve tener presente che gli interi sono rappresentati in little-endian, con il byte meno significativo al primo posto. Ad esempio il PIN 21664 che in esadecimale è `0x54a0` quando è rappresentato in 4 byte little-endian diventa `0xa0 0x54 0x00 0x00`. Per mandarlo sulla pipe si può usare il comando `echo` con le opzioni `-ne` che tolgono l'a-capo finale (-n) e interpretano le sequenze `\xa0` come byte (-a).

```
$ echo -ne "\xa0\x54\x00\x00" > tmpPipeInput
$ cat tmpPipeOutput
PIN errato
```

Possiamo anche inviare due PIN consecutivi

```
$ echo -ne "\xa0\x54\x00\x00\xa1\x54\x00\x00" > tmpPipeInput
$ cat tmpPipeOutput
PIN errato
PIN errato
```

Come si può notare crackme si aspetta i PIN codificati in 4 byte consecutivi senza separatore. Per inviarli da C **non serve fare alcuna conversione**: se il PIN è in una variabile intera sarà già rappresentato in little-endian.

Crackme con stringhe

La soluzione di inviare direttamente un intero nella sua rappresentazione interna può creare problemi di portabilità. Di solito si preferisce usare una rappresentazione che non dipenda dall'architettura o dal linguaggio utilizzato.

Questa variante di crackme legge i numeri interi come stringhe terminate da 0×00 e li converte, successivamente, in numeri interi per il confronto con il PIN.

Ecco il sorgente:

```c
#include <fcntl.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <stdio.h>

#define PNAME1 "tmpPipeInputChars"
#define PNAME2 "tmpPipeOutputChars"
#define SUCCESSO "PIN corretto. Sei autenticato\n"
#define FALLIMENTO "PIN errato\n"
#define PINsegreto 13495

void chiuditutto() {
    unlink(PNAME1);        // rimuove la pipe
    unlink(PNAME2);        // rimuove la pipe
    exit(1);
}

// stampa l'errore e termina
void die(char *s) {
    perror(s);
    exit(EXIT_FAILURE);
}

main() {
    int fdI,fdO,r,pin;
    char leggi[6];

    signal(SIGINT,chiuditutto);
```

```
29      signal(SIGINT,chiuditutto);
30
31      mkfifo(PNAME1,0666);    // crea la pipe, se esiste gia' non fa nulla
32      mkfifo(PNAME2,0666);    // crea la pipe, se esiste gia' non fa nulla
33
34      if ( (fdI = open (PNAME1,O_RDWR)) < 0 ) // apre la pipe per la lettura
35          die("errore apertura pipe\n");
36
37      if ( (fdO = open (PNAME2,O_RDWR)) < 0 ) // apre la pipe per la scrittura
38          die("errore apertura pipe\n");
39
40      while (1) {
41          r=0;
42
43          // legge il pin un carattere alla volta
44          while ( r<6 && read(fdI, &leggi[r], 1 ) && leggi[r] != 0 )
45              r++;
46
47          pin = atoi(leggi); // converte la stringa in intero
48
49          // Controlla la correttezza del PIN
50          if (pin == PINsegreto)
51              write(fdO, SUCCESSO, strlen(SUCCESSO)+1);
52              // qui si ha accesso alle risorse ...
53          else
54              write(fdO, FALLIMENTO, strlen(FALLIMENTO)+1);
55
56      }
57  }
```

Provare a scrivere il un programma crack-chars.c che interagisca con crackme-chars e scopra il PIN segreto.

NOTA: anche in questo caso si può interagire con il programma da linea di comando ma i PIN saranno stringhe terminate dal byte `0x00`:

```
$ echo -ne "21664\x00" > tmpPipeInputChars
$ cat tmpPipeOutputChars
PIN errato
^C
$ echo -ne "21664\x0021666\x00" > tmpPipeInputChars
$ cat tmpPipeOutputChars
PIN errato
PIN errato
```

In questo caso da C è necessario effettuare una conversione da interi a stringhe utilizzando, ad esempio, `sprintf` (vedere il manuale).

SOLUZIONE GAIA :

```c
#include <fcntl.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>

#define POUT "tmpPipeInput"
#define PIN "tmpPipeOutput"

void die(char *s) {
    perror(s);
    exit(EXIT_FAILURE);
}

int main() {
    int out;
    int in;
    int codice = 0;
    char leggi[100];

    mkfifo(POUT,0666);    // crea la pipe, se esiste gia' non fa nulla
    mkfifo(PIN,0666);    // crea la pipe, se esiste gia' non fa nulla

    if ( (out = open(POUT,O_RDWR)) < 0 )
        die("errore apertura pipe\n");

    if ( (in = open(PIN,O_RDWR)) < 0 )
        die("errore apertura pipe\n");

    while(codice < 99999){
        write(out, &codice, sizeof(int));

        int r=0;
        while ( r<100 && read(in, &leggi[r], 1 ) && leggi[r] != 0 )
            r++;

        if (leggi[4] == 'c'){
            printf("il pin corretto è:%d\n", codice);
            return 0;
         }
         codice++;
     }

     return 0;
 }
```