Interprocess communication: Pipe

- Affinché due processi possano cooperare, è spesso necessario che **comunichino** fra loro dei dati.
- Una prima possibile soluzione a questo problema consiste nell'utilizzo condiviso dei file (e.g., leggendo e scrivendo in un file comune). Tuttavia tale approccio risulta inefficiente; inoltre vi è la possibilità che si verifichino dei problemi di contesa della risorsa condivisa.
- UNIX, per risolvere il problema, mette a disposizione una primitiva, detta **pipe**, che consiste in un canale **unidirezionale** di comunicazione che collega un processo ad un altro, generalizzando il concetto di file.
- È possibile inviare dati in una pipe attraverso la system call `write` e leggere dalla pipe attraverso la system call `read`.

A livello della shell, la pipe

```
> ls -l | less
```

equivale alla sequenza di comandi seguente:

```
> ls -l > tmpfile
> less < tmpfile
> rm tmpfile
```

System call

Per creare una pipe, esiste l'apposita system call:

```
#include <unistd.h>

int pipe(int filedes[2]);
```

dove `filedes` è un array di due interi che conterrà i descrittori di file identificanti la pipe. Il primo (`filedes[0]`) serve a leggere dalla pipe, mentre il secondo (`filedes[1]`) serve a scrivervi.

La politica utilizzata da una pipe per gestire i messaggi è di tipo FIFO e non può essere cambiata.

L'utilità di una pipe diventa evidente quando è utilizzata in congiunzione alla chiamata di sistema `fork`, in quanto i descrittori di file rimangono aperti dopo la `fork` stessa.

Esempio (I)

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

#define MSGSIZE 16

char *msg1="hello, world #1";
char *msg2="hello, world #2";
char *msg3="hello, world #3";

main() {
  char inbuf[MSGSIZE];
  int p[2], j;
  pid_t pid;

  if(pipe(p)==-1) {
    perror("pipe call");
    exit(1);
  }
```

Esempio (II)

```
  switch(pid=fork()) {
    case -1:
      perror("fork call");
      exit(2);
    case 0:                      /* processo figlio */
      close(p[0]);               /* chiusura del descrittore di lettura */
      write(p[1], msg1, MSGSIZE);
      write(p[1], msg2, MSGSIZE);
      write(p[1], msg3, MSGSIZE);
      break;
    default:                     /* processo padre */
      close(p[1]);               /* chiusura del descrittore di scrittura */
      for(j=0; j<3; j++) {
        read(p[0], inbuf, MSGSIZE);
        printf("%s\n", inbuf);
      }
      wait(NULL);
  }
  exit(0);
}
```

Gioco guardie e ladri

Come esempio di utilizzo delle pipe si consideri il seguente caso:

- Si vuole scrivere un programma C per simulare il gioco "guardie e ladri":
  - il ladro (rappresentato dal carattere $) verrà mosso in modo casuale sullo schermo del terminale (80×24) dal computer;
  - la guardia (rappresentata dal carattere #) verrà mossa dall'utente tramite i tasti freccia;
  - il gioco terminerà quando la guardia ed il ladro si incontreranno.
- Nell'implementazione si useranno tre processi:
  - un processo padre, responsabile della visualizzazione e del controllo dell'evento in cui la guardia ed il ladro si incontrano;
  - un processo figlio che aggiorna la posizione del ladro;
  - un processo figlio che aggiorna la posizione della guardia.

Direttive di include, costanti e prototipi

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

#define PASSO     1 /* entita' dello spostamento del ladro */
#define SU       65 /* Freccia su */
#define GIU      66 /* Freccia giu */
#define SINISTRA 68 /* Freccia sinistra */
#define DESTRA   67 /* Freccia destra */
#define MAXX     80 /* Numero di colonne dello schermo */
#define MAXY     24 /* Numero di righe dello schermo */

/* Struttura per la comunicazione tra figli e padre */
struct pos {
  char c;           /* soggetto che invia il dato: ladro o guardia */
  int x;            /* coordinata x */
  int y;            /* coordinata y */
};

void ladro(int pipeout);
void guardia(int pipeout);
void controllo(int pipein);
```

Il main (I)

```
int main() {
  int filedes[2];
  int pid_ladro;
  int pid_guardia;

  initscr();   /* inizializzazione dello schermo */
  noecho();    /* i caratteri corrispondenti ai tasti premuti non saranno
                * visualizzati sullo schermo del terminale
                */
  curs_set(0); /* nasconde il cursore */

  if(pipe(filedes)==-1) {
    perror("Errore nella creazione della pipe.");
    exit(1);
  }
```

Il `main` (II)

```
  switch(pid_ladro=fork()) {
    case -1:
      perror("Errore nell'esecuzione della fork.");
      exit(1);
    case 0:                          /* processo figlio che gestisce il ladro */
      close(filedes[0]);             /* chiusura del descrittore di lettura */
      ladro(filedes[1]);
    default:                         /* processo padre */

      switch(pid_guardia=fork()) {
        case -1:
          perror("Errore nell'esecuzione della fork.");
          exit(1);
        case 0:                      /* processo figlio che gestisce la guardia */
          close(filedes[0]);         /* chiusura del descrittore di lettura */
          guardia(filedes[1]);
        default:                     /* processo padre */
          close(filedes[1]);         /* chiusura del descrittore di scrittura */
          controllo(filedes[0]);
      }

  }

  kill(pid_ladro,1);                 /* terminazione forzata del processo ladro */
  kill(pid_guardia,1);               /* terminazione forzata del processo guardia */
  endwin();                          /* ripristina il normale modo operativo del terminale */
  return 0;
}
```

La funzione per gestire la guardia

```
void guardia(int pipeout) {
  struct pos pos_guardia;
  pos_guardia.c='#';
  pos_guardia.x=MAXX-1;
  pos_guardia.y=MAXY-1;
  /* comunico la posizione iniziale della guardia: in basso a destra */
  write(pipeout,&pos_guardia,sizeof(pos_guardia));

  while(1) {
    char c;
    switch(c=getch()) {
      case SU:
        if(pos_guardia.y>0) pos_guardia.y-=1;
        break;
      case GIU:
        if(pos_guardia.y<MAXY-1) pos_guardia.y+=1;
        break;
      case SINISTRA:
        if(pos_guardia.x>0) pos_guardia.x-=1;
        break;
      case DESTRA:
        if(pos_guardia.x<MAXX-1) pos_guardia.x+=1;
        break;
    }
    /* comunico la nuova posizione al processo di controllo */
    write(pipeout,&pos_guardia,sizeof(pos_guardia));
  }
}
```

La funzione di controllo

```
void controllo (int pipein) {
  struct pos ladro, guardia, valore_letto;
  ladro.x=-1;
  guardia.x=-1;

  do {
    read(pipein,&valore_letto,sizeof(valore_letto));

    if(valore_letto.c=='$') {
      if (ladro.x>=0) { /* cancello la 'vecchia' posizione del ladro */
        mvaddch(ladro.y,ladro.x,' ');
      }
      ladro=valore_letto;
    }
    else {
      if (guardia.x>=0) { /* cancello la 'vecchia' posizione della guardia */
        mvaddch(guardia.y,guardia.x,' ');
      }
      guardia=valore_letto;
    }
    /* Disegno la nuova posizione */
    mvaddch(valore_letto.y,valore_letto.x,valore_letto.c);
    curs_set(0);
    refresh();
  } while (guardia.x!=ladro.x || guardia.y!=ladro.y);
}
```

Esercizio

Scrivere la funzione `ladro` che gestisce la posizione del ladro sullo schermo del terminale. Fare in modo che l'entità dello spostamento del ladro corrisponda a quanto specificato dalla costante simbolica `PASSO`.

Si faccia in modo inoltre che il ladro non esca dall'area 80×24 dello schermo durante i suoi spostamenti.

Suggerimento: si effettui una pausa tra uno spostamento e l'altro, per non rendere il movimento del ladro troppo "frenetico" sullo schermo (si utilizzi ad esempio la funzione `usleep`).

Per compilare il programma occorre usare l'opzione `-lncurses`:

```
gcc -lncurses -o guardieladri guardieladri.c
```