UNIVERSITÀ DEGLI STUDI DI PARMA
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica-Informatica-Telecomunicazioni
a.a. 2001-2002

CORSO DI SISTEMI OPERATIVI A - ESERCITAZIONE 5

1 Segnali

```
#include <signal.h>
void (*signal (int signal_name, void (*handler)(int)))
```

Il sistema operativo UNIX consente l'invio e la ricezione di segnali per la sincronizzazione tra processi. La funzione signal() viene utilizzata per definire il comportamento di un processo in seguito alla ricezione di uno specifico segnale (identificato dalla variabile signal_name). Il secondo parametro può assumere uno dei seguenti valori:

1. SIG_IGN se si vuole ignorare il segnale;
2. SIG_DFL se si vuole che il sistema operativo esegua l'azione di default del segnale;
3. un generico puntatore ad una funzione *handler* che ritorna un valore intero; tale funzione viene eseguita quando il processo riceve il segnale.

```
int kill (int process_id, int signal_name )
```

La funzione kill viene utilizzata per inviare un segnale ad un processo. Il parametro process_id rappresenta il PID del processo che viene segnalato, signal_name è il segnale che viene inviato al processo.

ELENCO DI ALCUNI SEGNALI

NOME	NUMERO	Descrizione
SIGINT	2	Interrupt da tastiera (Ctrl^c)
SIGKILL	9	Uccisione (non intercettabile o ignorabile)
SIGUSR1	10	Segnale utilizzabile liberamente
SIGUSR2	12	Segnale utilizzabile liberamente
SIGALRM	14	Allarme temporizzato
SIGTSTP	20	Sospensione da tastiera (Ctrl^z)

```
unsigned int alarm(int seconds)
```

Definisce un tempo in secondi dopo il quale il sistema operativo invia un segnale SIGALRM al processo chiamante.

```
int sigaction(int signum, const struct sigaction *act,struct sigaction *oldact);
```

La funzione sigaction() è la primitiva fondamentale per la gestione dei segnali affidabili.

```
int  sigprocmask(int  how,  const  sigset_t *set, sigset_t *oldset);
```

Un processo può eseminare e/o modificare la propria signal mask (insieme dei segnali che sta bloccando) attraverso la funzione sigprocmask().

```
int  sigsuspend(const sigset_t *mask);
```

La funzione sigsuspend() permette l'attivazione della maschera specificata e l'attesa in modo atomico.

```
#include <time.h>
int  nanosleep(const struct timespec *req, struct timespec *rem);
```

La funzione nanosleep() ritarda l'esecuzione del processo di un tempo specificato all'interno della struttura ∗req.

2 Pipe

```
int pipe (int file_descriptors[2])
```

Una possibile tecnica per la comunicazione tra processi è l'utilizzo della chiamata di sistema pipe(), che crea un canale di comunicazione. Il parametro file_descriptors[2] è un array che pipe() riempie con un file descriptor aperto in lettura, file_descriptor[0], e un file descriptor aperto in scrittura, file_descriptor[1].
Il buffer associato ad ogni pipe ha una dimensione finita (PIPE_BUF). Se un processo cerca di scrivere (write() system call) su una pipe il cui buffer è pieno il processo viene bloccato dal sistema operativo finchè il buffer non viene liberato attraverso una operazione di lettura (read() system call). Se il buffer è vuoto e un processo cerca di leggere da una pipe, il processo viene bloccato finchè non avviene una operazione di scrittura. Un processo padre può comunicare con un processo figlio attraverso una pipe in quanto il processo figlio possiede una copia dei file descriptor del padre.
Per convenzione le pipe vengono utilizzate come canali di comunicazione unidirezionali. Se due processi richiedono un canale di comunicazione bidirezionale tipicamente si creano due pipe.

Tutti i file con il codice sorgente degli esercizi proposti (es∗.c) si trovano nel direttorio:

/home/soa/eserc5/

3 Esercizi sui segnali

Esercizio 1:
In questo esempio il processo corrente definisce un handler per il segnale SIGINT. Provate a terminare il programma con il comando *Ctrl^c*. Per terminare effettivamente il programma sospendere l'esecuzione con il comando *Ctrl^z* e dalla shell invocare il comando *kill -9 <PID del programma sospeso>*. Per visualizzare l'elenco dei vostri processi utilizzare il comando *ps -elf | grep <user name>*.

```
#include <signal.h>

void catchint(int signo)
{
  printf("catchint: signo = %d\n", signo);
  /* non si prevedono azioni di terminazione: ritorno al segnalato */
}

int main()
{
  signal(SIGINT, catchint);
  for (;;)
    {
      printf("In attesa del segnale SIGINT (premere Ctrl^c)\n");
      sleep(1);
    }

  exit(0);
}
```

Esercizio 2:
Il processo padre crea un processo figlio e lo uccide.

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

int main()
{
  int pid;

  if ((pid=fork()) < 0)
    {
      perror("fork error");
      exit(1);
    }
  else
   if (pid == 0)
    {
      /* il figlio dorme fino a quando riceve il segnale SIGKILL, poi muore */
      for (;;)
       printf("sono il figlio e sto ciclando all'infinito!\n");
      printf("questo messaggio non dovrebbe mai essere visualizzato!\n");
      exit(0);
    }
  else
    {
      /* il padre invia un segnale SIGKILL al figlio */
      sleep(3);
      kill(pid, SIGKILL);
      printf("\nsono il padre e ho ucciso il figlio!!\n");
```

```
      exit(0);
    }
}
```

Modificare il programma in questo modo:

1. il processo padre invia dopo la system call sleep() un segnale SIGUSR1 al processo figlio (anzichè il segnale SIGKILL);
2. il processo figlio installa un gestore (handler) per il segnale SIGUSR1;
3. quando il figlio riceve il segnale SIGUSR1 stampa un messaggio di uscita e termina l'esecuzione.

Esercizio 3:
Esempio di un ping-pong di messaggi tra un processo padre e un processo figlio (versione con segnali non affidabili).

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

int ntimes = 0; /* variabile globale */

void catcher(int signo)
{
  printf("Processo %d ricevuto #%d volte\n", getpid(), ++ntimes);
}

int main()
{
  int pid, ppid;

  signal(SIGUSR1, catcher); /* il figlio la ereditera' */

  if ((pid = fork()) < 0)
    {
      perror("fork error");
      exit(1);
    }
  else if (pid == 0)
    {
      ppid = getppid();
      for (;;)
        {
          sleep(1);
          kill(ppid, SIGUSR1);
          pause();
        }
    }
  else
    {
```

```
      for (;;)
        {
          pause();
          sleep(1);
          kill(pid, SIGUSR1);
        }
    }
}
```

Esercizio 4:
Esempio di un ping-pong di messaggi tra un processo padre e un processo figlio (versione con segnali affidabili).

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

int ntimes = 0; /* variabile globale */

void catcher(int signo)
{
  printf("Processo %d ricevuto #%d volte\n", getpid(), ++ntimes);
}

int main()
{
  int pid, ppid;
  sigset_t set, oldset, zeromask;
  struct sigaction action, old_action;

  sigemptyset(&zeromask);
  /* azzera tutti i flag della maschera sa_mask nella struttura action */
  sigemptyset(&action.sa_mask);
  action.sa_handler = catcher;
  action.sa_flags = 0;

  sigemptyset(&set);
  sigaddset(&set, SIGUSR1);
  sigprocmask(SIG_BLOCK, &set, &oldset);

  /* assegna action per la gestione di SIGUSR1 */
  if (sigaction(SIGUSR1, &action, &old_action) == -1)
    perror("sigaction error");

  if ((pid=fork()) < 0)
    {
      perror("fork error");
      exit(1);
    }
  else
   if (pid == 0)
```

```
    {
      ppid = getppid();
      printf("figlio: mio padre e' %d\n", ppid);
      for (;;)
        {
          sleep(1);
          kill(ppid, SIGUSR1);
          sigsuspend(&zeromask);
        }
    }
  else
    {
      printf("padre: mio figlio e' %d\n", pid);
      for (;;)
        {
          sigsuspend(&zeromask);
          sleep(1);
          kill(pid, SIGUSR1);
        }
    }
}
```

Esercizio 5:
Esempio di utilizzo della system call alarm() per l'invio di un segnale temporizzato.

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

void handler(int signo)
{
  static int beeps = 0; /* static e' importante perche' ..*/

  printf("BEEP\n");
  if (++beeps < 5)
    alarm(1);
  else
    {
      printf("BOOM!\n");
      exit(0);
    }
}

int main()
{
  struct timespec ts;

  signal(SIGALRM, handler);
  alarm(1);

  ts.tv_sec = 20;
```

```
  ts.tv_nsec = 0;

  while(1)
    nanosleep (&ts, NULL);

  exit(0);
}
```

Esercizio 6:
Esempio di utilizzo di un segnale (SIGUSR1) per modificare il comportamento di un processo. Il programma ad intervalli di 1 secondo esegue un conteggio che viene invertito quando si riceve il segnale (SIGUSR1). Per inviare i segnali al processo eseguire il comando *kill -10 <PID del processo>* in un altro shell.

```
#include <signal.h>

int stato=1;
int i=0;

void handler(int signo)
{
  stato = !stato;
}

int main()
{
  sigset_t set;
  struct sigaction action, old_action;

  /* azzera tutti i flag della maschera sa_mask nella struttura action */
  sigemptyset(&action.sa_mask);
  action.sa_handler = handler;
  action.sa_flags = 0;

  /* assegna action per la gestione di SIGUSR1 */
  if (sigaction(SIGUSR1, &action, &old_action) == -1)
    perror("sigaction error");

  for(;;)
    {
      if (stato)
        printf("%d\n", i++);
      else
        printf("%d\n", i--);

      sleep(1);
    }

  exit(0);
}
```

4 Esercizi sulle pipe

Esercizio 7:
Esempio di comunicazione tra padre e figlio attraverso una pipe.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#define MSGSIZE 256

int main()
{
  int pid, j, c;
  int piped[2];
  char inpbuf[MSGSIZE];

  /*
     Apre la pipe creando due file descriptor,
     uno per la lettura e l'altro per la scrittura
     (vengono memorizzati nell'array piped[])
  */
  if (pipe(piped) < 0)
    exit(1);

  if ((pid = fork()) < 0)
    exit(2);
  else if (pid == 0)  /* figlio, che ha una copia di piped[] */
    {
      close(piped[1]);
      for (j = 1; j <= 10; j++)
        {
          read(piped[0], &c, sizeof (int));
          printf("Figlio: ho letto dalla pipe il numero %d\n", c);
        }
      exit(0);
    }
  else  /* padre */
    {
      close(piped[0]);
      for (j = 1; j <= 10; j++)
        {
          write(piped[1], &j, sizeof (int));
          sleep(1);
        }
      exit(0);
    }
}
```

Modificare il programma in modo che stampi la lunghezza del buffer della pipe (occorre includere il file *limits.h*).

Esercizio 8:
Il padre crea due figli, ciascuno dei quali gli invia un numero (utilizzando la pipe individuata dal descrittore pdchf). Dopodichè il padre invia a ciascun figlio (utilizzando due ulteriori pipe: pdfch1 e pdfch2) un numero casuale compreso tra 1 e il numero precedentemente ricevuto.

```
#include <stdio.h>
#include <stdlib.h>  /* serve per rand() e srand()*/

int main(int argc , char* argv[])
{
  int pdchf[2], pdfch1[2], pdfch2[2];
  int pid1, pid2;

  struct msg {
    int mpid;
    int n;
  } m1, m2;

  int n1, n2;

  /* Apre la pipe per la comunicazione da figli a padre */
  if (pipe(pdchf) < 0)
    exit(1);

  /* Apre la pipe per la comunicazione da padre a figlio 1*/
  if (pipe(pdfch1) < 0)
    exit(1);

  /* Apre la pipe per la comunicazione da padre a figlio 2*/
  if (pipe(pdfch2) < 0)
    exit(1);

  if ((pid1 = fork()) < 0)
    exit(2);
  else if (pid1 == 0)  /* figlio 1 */
    {
      close(pdchf[0]);   /* sulla pipe chf il figlio 1 deve solo scrivere */
      close(pdfch1[1]);  /* sulla pipe fch1 il figlio 1 deve solo leggere */
      close(pdfch2[0]);  /* sulla pipe fch2 il figlio 1 non deve ne' leggere */
      close(pdfch2[1]);  /* ne' scrivere */
      m1.mpid = getpid();
      m1.n = atoi(argv[1]);
      /* ora comunica al padre il proprio messaggio */
      write(pdchf[1], &m1, sizeof(m1));
      printf("figlio 1: ho scritto al padre %d\n", m1.n);
```

```
    read(pdfch1[0], &m1, sizeof(m1));  /* legge il messaggio del padre */
    printf("figlio 1: il padre mi ha scritto %d\n", m1.n);
    exit(0);
  }
else  /* padre */
  {
    if ((pid2 = fork()) < 0)
      exit(2);
    else if (pid2 == 0)  /* figlio 2 */
      {
        close(pdchf[0]);   /* sulla pipe chf il figlio 2 deve solo scrivere */
        close(pdfch2[1]);  /* sulla pipe fch2 il figlio 2 deve solo leggere */
        close(pdfch1[0]);  /* sulla pipe fch1 il figlio 2 non deve ne' leggere */
        close(pdfch1[1]);  /* ne' scrivere */
        m2.mpid = getpid();
        m2.n = atoi(argv[2]);
        /* ora comunica al padre il proprio messaggio */
        write(pdchf[1], &m2, sizeof(m2));
        printf("figlio 2: ho scritto al padre %d\n", m2.n);
        read(pdfch2[0], &m2, sizeof(m2));  /* legge il messaggio del padre */
        printf("figlio 2: il padre mi ha scritto %d\n", m2.n);
        exit(0);
      }
    else  /* padre */
      {
        close(pdchf[1]);   /* sulla pipe chf il padre deve solo leggere */
        close(pdfch1[0]);  /* sulla pipe fch1 il padre deve solo scrivere */
        close(pdfch2[0]);  /* sulla pipe fch2 il padre deve solo scrivere */

        read(pdchf[0], &m1, sizeof(m1));
        read(pdchf[0], &m2, sizeof(m2));

        /* ora genera un nuovo seme per la successiva sequenza di chiamate a rand()
        srand(time(0));
        /* genera 2 numeri casuali compresi tra 1 e nt
           e li assegna rispettivamente a n1 e a n2    */
        n1 = 1+(int) (m1.n * (rand()/(RAND_MAX+1.0)));
        n2 = 1+(int) (m2.n * (rand()/(RAND_MAX+1.0)));

        m1.n = n1;
        m2.n = n2;

        if (m1.mpid == pid1)
          {
            write(pdfch1[1], &m1, sizeof(m1));
            write(pdfch2[1], &m2, sizeof(m2));
          }
        else
```

```
            {
              write(pdfch1[1], &m2, sizeof(m2));
              write(pdfch2[1], &m1, sizeof(m1));
            }
          sleep(1);
          exit(0);
        }
    }
}
```

Esercizio 9:
Esempio di utilizzo di una pipe tra processi per la realizzazione di uno pseudo comando pipe.
La sintassi per l'esecuzione del programma è la seguente:
./pipe2 comando1 ! comando2
(per esempio *./pipe2 ls ! sort*)

```
#include <stdio.h>

int join(char* com1[], char *com2[])
{
  int status, pid;
  int piped[2];

  switch(fork())
    {
      case -1:  /* errore */  return 1;
      case  0:  /* figlio */  break;  /* esce dal case */
      default:  /* padre: attende il figlio */  wait(&status); return status;
    }

  /* il figlio apre la pipe e poi crea un suo figlio (nipote del primo processo)*/
  if (pipe(piped) < 0)
    return 2;
  if ((pid = fork()) < 0)
    return 3;
  else if (pid == 0)
    {
      close(0);
      dup(piped[0]);
      close(piped[0]);
      close(piped[1]);
      execvp(com2[0], com2);
      return 4;
    }
  else
    {
      close(1);
      dup(piped[1]);
      close(piped[0]);
      close(piped[1]);
```

```
      execvp(com1[0], com1);
      return 5;
    }
}

int main(int argc, char **argv)
{
  int integi, i, j;
  char *temp1[10], *temp2[10];

  /*
     si devono fornire nella linea di comando due comandi distinti,
     separati dal carattere ! (non usare | perchè questo viene direttamente
     interpretato dallo Shell come una pipe)
  */

  if (argc > 2)
    {
      for (i = 1; (i < argc) && ((strcmp(argv[i], "!"))!=0); i++)
        temp1[i-1] = argv[i];
      temp1[i-1] = (char *)0;
      i++;
      for (j = 1; i < argc; i++, j++)
        temp2[j-1]=argv[i];
      temp2[j-1] = (char *)0;
    }
  else
    {
      printf("errore");
      exit(-2);
    }
  integi = join(temp1, temp2);
  exit(integi);
}
```

Esercizio 10 (da completare):
Un processo padre crea N processi figli. Ciascun processo figlio è identificato da una variabile intera id (id=0,1,2,3...,N-1 con N pari).

1. Se argv[1]='a' ogni processo figlio con id pari manda un segnale (SIGUSR1) al processo id+1;
2. Se argv[1]='b' ogni processo figlio con $id < N/2$ manda un segnale (SIGUSR1) al processo $id + N/2$.

Completare il programma scrivendo il corpo della funzione body_proc() che viene eseguita da ciascun figlio.

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

```
#define N 10

int pg[2];
int tabpid[N];
char arg1;

void handler(int signo)
{
  printf("Sono il processo %d e ho ricevuto il segnale %d\n",getpid(),signo);
}

void body_proc(int id)
{
}

main (int argc, char* argv[])
{
  int i;

  if (pipe(pg)<0)
    {
      perror("pipe error");
      exit(-1);
    }
  arg1= argv[1][0]; /* primo carattere del secondo argomento */
  for (i=0;i<N;i++)
    {
      if ((tabpid[i]=fork())<0)
        {
          perror("fork error");
          exit(-1);
        }
      else
        if (tabpid[i]==0) body_proc(i);
    }
  printf("Sono il padre e scrivo sulla pipe la tabella dei pid\n");
  write(pg[1],tabpid,sizeof tabpid);

  return 0;
}
```