

CORSO DI SISTEMI OPERATIVI A - ESERCITAZIONE 5

1 Segnali

Vi sono spesso eventi importanti da notificare ai processi:

- tasti speciali sul terminale (es. Ctrl^c)
- eccezioni hardware (es. divisione per 0)
- primitiva/comando kill (es. *kill -9 1221*)
- condizioni software (es. scadenza di un timer)

L'arrivo di tali **eventi asincroni** può richiedere un'immediata gestione da parte del processo. Il sistema operativo UNIX consente l'invio e la ricezione di segnali per la sincronizzazione tra processi con due principali modalità principali : una gestione tradizionale e limitata, detta inaffidabile perchè può causare la perdita di segnali inviati, e una più sofisticata, detta affidabile.

L'interfaccia semplificata **signal** è definita nello standard ANSI C e in alcune versioni di UNIX può contribuire ad una **gestione inaffidabile dei segnali**, ed è quindi preferibile l'uso della sigaction (vedi più avanti).

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

La funzione *signal* viene utilizzata per definire il comportamento di un processo in seguito alla ricezione di uno specifico segnale (identificato dall'argomento signumber). Il secondo parametro può assumere uno dei seguenti valori:

1. SIG_IGN se si vuole ignorare il segnale (possibile solo per alcuni tipi di 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.

```
#include <signal.h>
int kill (int process_id, int signumber )
```

La funzione *kill* viene utilizzata per inviare un segnale ad un processo. Il parametro *process_id* rappresenta il PID del processo che viene segnalato, *signumber* è il segnale che viene inviato al processo, normalmente designato mediante una costante che è definita in <signal.h>. Si consiglia l'utilizzo di una costante (ad es. SIGUSR1) piuttosto di un valore numerico esplicito al fine di ottenere codice portabile: ad esempio il valore numerico associato a SIGUSR1 è 10 in Linux, 16 in Solaris e 30 nell'ambiente POSIX Cygwin per Windows.

NOME	NUMERO	Descrizione
SIGINT	2	Interrupt da tastiera (Ctrl^c)
SIGKILL	4	Uccisione (non intercettabile o ignorabile)
SIGALRM	14	Allarme temporizzato
SIGTSTP	18	Sospensione da tastiera (Ctrl^z)
SIGUSR1	10	Segnale utilizzabile liberamente
SIGUSR2	12	Segnale utilizzabile liberamente

Tabella 1: Elenco di alcuni segnali.

```
#include <unistd.h>
unsigned int alarm(int seconds)
```

La funzione *alarm* definisce un tempo in secondi dopo il quale il sistema operativo invia un segnale SIGALRM al processo chiamante (in inglese “alarm” vuol dire “sveglia”).

```
#include <unistd.h>
unsigned int pause(void)
```

La funzione *pause* mette in pausa il processo fino alla ricezione di un segnale.

```
#include <unistd.h>
unsigned int sleep(unsigned int n_seconds);
```

La funzione *sleep* mette in pausa il processo di *n_seconds* secondi. Il processo si può risvegliare prima di *n_seconds* secondi se nel frattempo riceve un segnale non ignorato.

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

La funzione *nanosleep* ritarda l'esecuzione del processo di un tempo specificato da **req*, che è un puntatore ad una struttura di questo tipo:

```
struct timespec
{
    time_t    tv_sec;           /* seconds */
    long      tv_nsec;         /* nanoseconds */
};
```

Nel caso di arrivo di un segnale, la *nanosleep* interrompe l'attesa, ritorna con -1, assegna ad *errno* il valore EINTR, e restituisce il tempo rimanente nel parametro **rem*.

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

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

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

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

2 Pipe

```
#include <unistd.h>
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.

3 Esercizi sui segnali

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

`/home/soa/eserc5/`

Esercizio 1:

In questo esempio il processo corrente definisce un handler (funzione di gestione di un segnale) per il segnale `SIGINT`. Provate a terminare il programma con il comando `Ctrl^c`. Dando il comando `Ctrl^c` si invia ai processi in corso (in foreground) un segnale `SIGINT`, che però in questo caso è catturato dall'handler. 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 processi utilizzare il comando `ps`. (Per visualizzare l'elenco completo dei vostri processi utilizzare il comando `ps -elf | grep <user name>`)

```
#include <signal.h>    //signal
#include <stdio.h>      //printf
#include <unistd.h>     //sleep
#include <stdlib.h>     //exit

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);
}

```

Modificare il programma in modo che catturi il segnale SIGTSTP, generato dall'utente premendo *Ctrl^z*.

Esercizio 2:

Il processo padre crea un processo figlio e lo uccide.

```

#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int pid;

    if ((pid=fork()) < 0)
    {
        perror("fork error");
        exit(1);
    }
    else
    if (pid == 0)
    {
        /* il figlio attende in un ciclo con \textit{busy waiting} fino a quando riceve
        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 visualizza 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>
#include <stdio.h>
#include <stdlib.h>

/* variabile globale contatore dei segnali SIGUSR1
   ricevuti da ciascun processo */

int ntimes = 0;

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

int main()
{
    int pid, ppid;

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

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

```

```

else if (pid == 0)
{
    ppid = getppid();
    for (;;)
    {
        printf("figlio: mio padre e' %d\n", ppid);
        sleep(1);
        kill(ppid, SIGUSR1);
        pause();
    }
}
else
{
    for (;;)
    {
        printf("padre: mio figlio e' %d\n", pid);
        pause();
        sleep(1);
        kill(pid, SIGUSR1);
    }
}
}

```

Osservazione: per quale ragione è necessaria la chiamata alla funzione `pause`? Provare a rimuoverla e vedere come si comporta il programma. Il comportamento apparentemente anomalo è dovuto alle caratteristiche della funzione `sleep` (controllatene la descrizione), che fa risvegliare il processo prima che siano passati `n.seconds` se questi riceve un segnale non ignorato. La stessa cosa vale anche per la funzione `nanosleep`.

Facoltativo: Ripristinare la pause e modificare il programma in modo che il padre ad ogni ciclo mandi un numero di segnali al figlio pari a $(ntimes+1)$. Come vi aspettate si comporti il figlio in risposta a questi segnali? Provate a mettere in atto questa modifica, e controllate se la vostra previsione è corretta.

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>
#include <stdio.h>
#include <stdlib.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, zeromask;
    struct sigaction action;

    sigemptyset(&zeromask);
    /* prepara una maschera di segnali vuota (da usare più avanti) */

    sigemptyset(&action.sa_mask);
    /* azzera tutti i flag della maschera di segnali sa_mask,
       presente all'interno della struttura action.
       Tale maschera corrisponde all'insieme di segnali
       che saranno bloccati durante l'esecuzione della procedura di
       gestione del segnale a cui action verrà associata */

    action.sa_handler = catcher; /* assegna la funzione da chiamare
                                   alla ricezione del segnale */
    action.sa_flags = 0;

    //PROTEGGO IL PROCESSO DALL'ARRIVO DI SEGNALI SIGUSR1
    //AL DI FUORI DEL MOMENTO IN CUI LO ATTENDO
    //Il "momento in cui lo attendo" corrisponde al punto in cui pongo la sigsuspend.
    sigemptyset(&set); /* azzera la maschera di segnali set... */
    sigaddset(&set, SIGUSR1); /* ...e quindi ci aggiunge SIGUSR1 */
    sigprocmask(SIG_BLOCK, &set, NULL); /* aggiunge (in OR) all'insieme
    dei segnali mascherati dal processo
        corrente quelli contenuti in set */

    if (sigaction(SIGUSR1, &action, NULL) == -1)
    /* assegna action per la gestione di SIGUSR1 */

        perror("sigaction error");

    /* da questo punto in poi, alla ricezione di un segnale SIGUSR1
       il processo corrente (come anche tutti i suoi figli)
       risponderà con l'invocazione della funzione catcher().
       In questo momento però i segnali SIGUSR1 sono invisibili al
       processo grazie alla precedente invocazione della sigprocmask:
       grazie alla sigsuspend(&zeromask) sblocco i segnali nel momento
       in cui sono pronto a riceverli.
       */
}
```

```

if ((pid=fork()) < 0)
{
    perror("fork error");
    exit(1);
}
else
if (pid == 0) /* siamo nel figlio */
{
    ppid = getppid();
    printf("figlio: mio padre e' %d\n", ppid);
    for (;;) /* ciclo infinito */
    {
        sleep(1);
        kill(ppid, SIGUSR1); /* inviamo al padre un segnale SIGUSR1... */
        sigsuspend(&zeromask); /* ...e ci sospendiamo fino alla ricezione
                                di un segnale qualunque*/
    }
}
else /* siamo nel padre */
{
    printf("padre: mio figlio e' %d\n", pid);
    for (;;)
    {
        sigsuspend(&zeromask); /* attendiamo la ricezione di un segnale... */
        sleep(1);
        kill(pid, SIGUSR1); /* ...e quindi ne inviamo uno di tipo
                             SIGUSR1 al nostro figlio */
    }
}
}

```

La gestione affidabile dei segnali ha una struttura un po' complicata, che però è sempre la stessa.

Nel fare queste operazioni si utilizzano delle maschere (tipo di dato `sigset_t`), ovvero un insieme di bit, ognuno associato ad un tipo di segnale, che possono essere accesi o spenti. Per accendere o spegnere questi bit si utilizzano le funzioni `sigemptyset`, `sigaddset`, `sigfillset` ecc. (vederne il manuale). Con

```
sigemptyset(&zeromask);
```

per esempio si crea una maschera di nome “zeromask” (di cui faremo uso più avanti) tutta azzerata.

Per preparare l'azione del processo all'arrivo di un determinato segnale (in questo caso `SIGUSR1`) si prepara una struttura di tipo `sigaction`. Il primo campo della struttura da sistemare è la maschera `sa_mask` (di tipo `sigset_t`), che va semplicemente vuotato se non vogliamo proteggere il processo dall'arrivo di altri segnali durante l'esecuzione dell'handler (gestore del segnale) (*man 2 sigaction* per approfondimenti).

```
sigemptyset(&action.sa_mask);
```


Poi si indica nel campo `sa_handler` la funzione da eseguire all'arrivo del segnale, ovvero l'handler definito più in alto nel codice.

```
action.sa_handler = catcher;
```

Il campo `sa_flags` va solitamente posto a zero. (*man 2 sigaction* per vedere i possibili valori assegnabili)

```
action.sa_flags = 0;
```

Infine l'azione va "comunicata" al sistema operativo, indicando a quale segnale associare l'azione. La condizione sul valore di ritorno della `sigaction` serve a verificare che l'operazione sia andata a buon fine.

```
if (sigaction(SIGUSR1, &action, NULL) == -1)
```

Ci sono casi in cui è necessario per certe fasi del processo bloccare l'arrivo dei segnali, per poi riattivarlo quando si è pronti alla ricezione. Per specificare quali segnali non devono arrivare al processo (e rimanere pendenti), si procede preparando una maschera di segnali vuota...

```
sigemptyset(&set);
```

... si aggiunge a questa maschera il segnale (o i segnali, con più `sigaddset`) su cui si vuole imporre la condizione ...

```
sigaddset(&set, SIGUSR1);
```

... e si chiede al sistema operativo di aggiungere ("OR" bit a bit) questa maschera all'attuale maschera dei segnali bloccati (opzione `SIG_BLOCK`):

```
sigprocmask(SIG_BLOCK, &set, NULL);
```

Per ripristinare la ricezione del segnale è sufficiente utilizzare la stessa maschera, stavolta però con l'opzione `SIG_UNBLOCK`.

```
sigprocmask(SIG_UNBLOCK, &set, NULL);
```

In alternativa, per sbloccare TUTTI i segnali si può utilizzare la `zeromask` creata in precedenza e l'opzione `SIG_SETMASK` (per sovrascrivere l'attuale maschera dei segnali bloccati):

```
sigprocmask(SIG_SETMASK, &zeromask, NULL);
```

Se il blocco di un segnale è attuato in previsione di una `sigsuspend` (cioè di un punto del codice in cui il processo dovrà sospendersi in attesa dell'arrivo di un segnale) non è necessario sbloccare i segnali ma è sufficiente attuare una `sigsuspend` con maschera vuota:

```
sigsuspend(&zeromask);
```

Infatti la maschera passata come parametro alla `sigsuspend` indica quali segnali devono essere bloccati durante la `sigsuspend` (in questo caso nessuno). **NOTA:** Se sono presenti segnali pendenti, bloccati dalla precedente `sigprocmask`, la `sigsuspend(&zeromask)` ritorna immediatamente.

Esercizio 5:

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

```

#include <signal.h>
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.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);
}

```

Provare a vedere qual è il comportamento del programma togliendo static. Per capire il funzionamento anomalo, visualizzare la variabile beeps all'interno dell'handler.

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'altra shell.

```

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

```

```

#include <unistd.h>

int stato=1, i=0;

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

int main()
{
    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 <stdlib.h>
#include <unistd.h>

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

```

```

/*
    Crea la pipe ottenendo 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]); /* nel figlio il fd per la scrittura non serve */
    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]); /* nel padre il fd per la lettura non serve */
    for (j = 1; j <= 10; j++)
    {
        write(piped[1], &j, sizeof (int)); /* l'intero che viene inviato come
messaggio e' in questo caso il
valore corrente della variabile j */
        sleep(1);
    }
    exit(0);
}
}

```

Modificare il programma in modo che visualizzi 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. La sintassi per l'esecuzione del programma è *nomeEseguibile numero1 numero2*

```

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

```

```

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 :
non gli serve quindi il descrittore per la lettura dalla pipe */

        close(pdfch1[1]); /* sulla pipe fch1 il figlio 1 deve solo leggere :
non gli serve quindi il descrittore per la scrittura sulla pipe*/

        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)));

        if (m1.mpid == pid1)
        {
            m1.n = n1;
            m2.n = n2;
            write(pdfch1[1], &m1, sizeof(m1));
            write(pdfch2[1], &m2, sizeof(m2));
        }
        else
        {
            m1.n = n2;
            m2.n = n1;
            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 che realizza il *piping* dei comandi offerto dagli shell UNIX. La sintassi per l'esecuzione del programma è la seguente:

./pipe2 comando1 ! comando2

(per esempio *./pipe2 ls ! sort*)

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.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); /* ora il fd 0 (stdin) è libero */
        dup(piped[0]); /* l'fd 0 (stdin) diventa un sinonimo di piped [0] */
        close(piped[0]);
        close(piped[1]); /* abbiamo liberato i due valori di fd salvati in piped,
                           dato che non li useremo più */
        execvp(com2[0], com2); /* esegue il programma specificato in com2, utilizzando
                                come input quello proveniente dallo stdin,
                                ora associato all'ingresso della pipe */

        return 4;
    }
}

```

```

    }
else
{
    close(1); /* ora il fd 1 (stdout) è libero */
    dup(piped[1]); /* l'fd 1 (stdout) diventa un sinonimo di piped [1] */
    close(piped[0]);
    close(piped[1]); /* abbiamo liberato i due valori di fd salvati in piped,
                       dato che non li useremo più */
    execvp(com1[0], com1); /* esegue il programma specificato in com1,
                           inviando l'output allo stdout,
                           ora associato all'uscita della pipe */

    return 5;
}
}

```

```

int main(int argc, char **argv)
{
    int integri, 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);
    }
    integri = join(temp1, temp2);
    exit(integi);
}

```

Esercizio 10:

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 $id + N/2$.

```
#include <stdio.h>
#include <ctype.h>
#include <signal.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.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)
{
}

int 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;  
}
```

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