Pause e Alarm

Un'altra funzione utile è pause() che sospende il processo chiamante sino alla ricezione di un segnale e risulta utile per sincronizzazioni interprocesso basate su segnali.

```
int pause(void);
```

La funzione alarm() permette di lanciare un segnale di tipo SIGALRM ad un determinato processo dopo un certo numero di secondi, ritorna il numero di secondi mancanti all'invio del segnale.

```
unsigned int alarm (unsigned int secs);
```

Esempio alarm.c

```
#include <unistd.h>
#include<signal.h>
void announce();
int main (int argc, char*argv[])
 {if(argc!=2)
   {printf(”Uso: %s secondi\n”,argv[0]);exit(1);}
  signal(SIGALRM,announce);
  alarm((unsigned)atoi(argv[1]));
  pause();} /* attende un segnale */
void announce()
{ fprintf(stdout,”Sveglia! \n”);
  exit (0);}
```

Bloccare i Segnali

Bloccare un segnale significa sostanzialmente lasciare il segnale pendente ad arbitrio del programmatore.

In genere si blocca il segnale quando un programma esegue un gruppo di istruzioni critiche e lo si sblocca immediatamente dopo. La libreria GNU C mette a disposizione del programmatore delle funzioni in grado di gestire il bloccaggio e lo sbloccaggio dei segnali.

Esse utilizzano un particolare tipo di dato al fine di definire quali segnali debbano essere bloccati: sigset_t, implementato sia come un intero che come una struttura e definito in signal.h.

L'inizializzazione di una variabile di questo tipo deve essere fatto in uno dei seguenti modi:

- inizializzato vuoto attraverso la funzione

  int sigemptyset(sigset_t *SET)

  in seguito si aggiungono singolarmente i segnali che devono far parte dell'insieme.

- inizializzato pieno attraverso la funzione

  int sigfillset(sigset_t *SET)

  in seguito si tolgono singolarmente quei segnali che non devono far parte dell'insieme.

L'insieme dei segnali correntemente bloccati, durante l'esecuzione di un processo, prende il nome di signal mask ed ogni processo ne ha uno.

Ogni processo figlio eredita dal padre la propria signal mask, ogni processo può comunque intervenire su di essa, modificandola, attraverso l'uso delle funzioni:

- int sigaddset (sigset_t *SET, int SIGNUM)
  per aggiungere il segnale *SIGNUM* al *signal set* puntato da *SET*. Ritorna 0 in caso di successo e -1 in caso di fallimento. Il valore della variabile *errno* può assumere il valore *EINVAL* nel caso si tenti di passare alla funzione un segnale non valido.
- int sigdelset (sigset_t *SET, int SIGNUM)
  per elimina il segnale SIGNUM dal *signal set* puntato da SET. Per il resto valgono le stesse caratteristiche della funzione precedentemente trattata.

- int sigismember (const sigset_t *SET, int SIGNUM)
  Come facilmente intuibile questa è una funzione di test. Viene infatti verificato che il segnale SIGNUM appartenga al signal set puntato da SET. In caso affermativo viene restituito il valore 1 altrimenti, in caso negativo, 0. Qualora si verificasse un errore il valore restituito è -1. La variabile errno può assumere il valore EINVAL nel caso si tenti di passare alla funzione un segnale non valido.
- int sigprocmask (int HOW, const sigset_t *restrict SET, sigset_t *restrict OLDSET)
  Questa funzione è utilizzata, per la manipolazione della *signal mask* , e l'effettivo comportamento di questa chiamata è determinato dalla variabile *HOW*.

Se *HOW=SIG_BLOCK* l'insieme dei segnali definito in *SET* viene aggiunto all'insieme dei segnali della *signal mask* corrente.
Se *HOW=SIG_UNBLOCK* l'insieme dei segnali definito in *SET* viene rimosso dalla *signal mask*.
Se *HOW=SIG_SETMASK* l'insieme dei segnali definito in *SET* viene utilizzato come nuova *signal mask*, la variabile *OLDSET* è utilizzata per tenere traccia della *signal mask* precedente alla modifica, qualora si volesse, ad esempio, tornare a riutilizzarla. In caso di successo la funzione ritorna il valore 0, -1 in caso di fallimento.

- int sigpending (sigset_t *SET)

Questa funzione è utilizzata per conoscere quali segnali sono pendenti in un determinato momento. Le informazioni relative

vengono memorizzate in SET. Il valore 0 viene ritornato in caso di successo, -1 in caso di errore.

Infine la funzione sigsuspend la cui sintassi è la seguente

int sigsuspend(sigset_t *SET)

rimpiazza la *signal mask* con la variabile *SET*, bloccando quindi tutti i segnali in esso definiti.

Il processo viene sospeso fino all'arrivo di un segnale che non fa parte di *SET*. In seguito all'avvenuta ricezione di questo segnale viene eseguito l'eventuale signal handler e la funzione ritorna ripristinando la *signal mask* precedente alla sua chiamata.

Consideriamo come esempio il seguente programma che durante l'esecuzione del signal handler, determinati segnali vengono bloccati.

Si procede in modo che dopo l'esecuzione del programma viene inviato il segnale SIGUSR1 al processo con il comando

kill -s SIGUSR1 <pid>.

Se SIGUSR2 viene inviato prima di digitare invio all'interno del signal handler per SIGUSR1, esso viene sospeso e bloccato, dopo il primo invio verrà sbloccato, e la ricezione di SIGUSR2 dopo il primo invio viene invece gestita immediatamente.

Il blocco di SIGUSR2 si ottiene tramite le funzioni sigemptyset, sigaddset, e sigprocmask.

usr_signal2.c

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

void sig_user_print1(int signum);
void sig_user_print2(int signum);
int main(int argc, char *argv[]);

void sig_user_print1(int signum)
{sigset_t block_sig;
 sigemptyset(&block_sig);// inizializza l'insieme vuoto dei
segnali bloccati
 sigaddset(&block_sig,SIGUSR2);// aggiungi SIGUSR2
 sigprocmask(SIG_BLOCK,&block_sig,NULL);// blocca SIGUSR2
 printf("SIGUSR2 blocked in SIGUSR1\n");// avvisa della ricezione
di SIGUSR1
 printf("SIGUSR1 sent...\n");
 printf("Press enter\n");
 getchar();// attendi invio
 printf("SIGUSR2 unlocked...\n");
```

```
 sigprocmask(SIG_UNBLOCK,&block_sig,NULL);// sblocca SIGUSR2
 getchar();// attendi invio
 printf("Exit SIGUSR1\n");
}

void sig_user_print2(int signum)
{printf("SIGUSR2 sent...\n");} // avvisa dell'invio del segnale

int main(int argc, char *argv[])
{// puntatori alle funzioni dei signal handler
 void (*sig_old1)(int);
 void (*sig_old2)(int);
 printf("My pid is: %d\n",getpid());
 // ottieni il pid da utilizzare col comando kill
 // sostituisci i signal handler
 sig_old1=signal(SIGUSR1,sig_user_print1);
 sig_old2=signal(SIGUSR2,sig_user_print2);
 printf("Wait or type enter to exit...\n");
 getchar();// attendi un carattere (invio)
 // ripristina i signal handler originari
 signal(SIGUSR1,sig_old1);
 signal(SIGUSR2,sig_old2);
 exit(EXIT_SUCCESS);}
```

In questo secondo esempio SIGUSR2 viene bloccato sia nel corpo del programma che nel signal handler SIGUSR1.
La maschera di blocco del signal handler vale solo all'interno del signal handler, al ritorno da esso viene ripristinata la maschera di blocco all'interno del programma.
Infatti anche dopo l'uscita dall'handler dopo la ricezione di SIGUSR1 seguito da due volte invio, pur avendo effettuato lo sblocco, la ricezione nel main di SIGUSR2 viene bloccata.
Quindi è necessario lo sblocco successivo nel main dopo un ulteriore l'invio.

usr_signal3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
void sig_user_print1(int signum);
void sig_user_print2(int signum);
```

```c
int main(int argc, char *argv[]);
void sig_user_print1(int signum)
{sigset_t block_sig;
 // inizializza l'insieme vuoto dei segnali bloccati
 sigemptyset(&block_sig); // aggiungi SIGUSR2
 sigaddset(&block_sig,SIGUSR2); // blocca SIGUSR2
 sigprocmask(SIG_BLOCK,&block_sig,NULL);
 printf("SIGUSR2 blocked in SIGUSR1\n");// avvisa invio SIGUSR1
 printf("SIGUSR1 sent...\n");printf("Press enter\n");
 getchar();// attendi invio
 printf("SIGUSR2 unblocked in SIGUSR1\n");
 sigprocmask(SIG_UNBLOCK,&block_sig,NULL); // sblocca SIGUSR2
 printf("Press enter\n");
 getchar();// attendi invio
 printf("Exit SIGUSR1\n");}

void sig_user_print2(int signum)
{// mostra l'avviso di ricezione
 printf("SIGUSR2 sent...\n");}

int main(int argc, char *argv[])
{// puntatori alle funzioni di signal handler
 void (*sig_old1)(int);
 void (*sig_old2)(int);
```

```
 sigset_t block_sig;
 // inizializza l'insieme vuoto dei segnali bloccati
 sigemptyset(&block_sig);
 sigaddset(&block_sig,SIGUSR2); // aggiungi SIGUSR2

 printf("My pid is: %d\n",getpid());// pid da utilizzare con kill
 sigprocmask(SIG_BLOCK,&block_sig,NULL); // blocca SIGUSR2
 printf("SIGUSR2 blocked outside SIGUSR1\n");
 // modifica i signal handler
 sig_old1=signal(SIGUSR1,sig_user_print1);
 sig_old2=signal(SIGUSR2,sig_user_print2);
 printf("Wait or type enter to continue...\n");
 getchar();// attendi invio
 printf("SIGUSR2 unblocked outside SIGUSR1\n");
 sigprocmask(SIG_UNBLOCK,&block_sig,NULL); // sblocca SIGUSR2
 printf("Wait or type enter to exit...\n");
 getchar();  // attendi invio
 printf("Exit...\n");
 // ripristina i signal handler di default
 signal(SIGUSR1,sig_old1);
 signal(SIGUSR2,sig_old2);
 exit(EXIT_SUCCESS);
}
```

In questo ultimo esempio utilizziamo il segnale SIGCHLD per gestire la terminazione dei processi figli con relativo rilascio delle risorse.
Il padre genera dei processi figli che terminano in un tempo casuale differente e attende la loro terminazione.
All'arrivo di SIGCHLD il padre controlla se ci sono figli che hanno terminato e rilascia le sue risorse.
Poiché è possibile che mentre è in esecuzione l'handler di SIGCHLD terminano più di un figlio, e solo un SIGCHLD risulta pendente è necessario un ciclo while.
Per comunicare tra processi tramite variabili esse devono essere di tipo volatile.
Il qualificatore volatile serve segnalare al compilatore che la variabile può essere modificata in modo non deterministico, quindi non prevedibile in base al programma, da qualcosa di

esterno, come per esempio un altro processo, il sistema operativo, o un dispositivo hardware.
Se ad esempio in un programma si utilizza un ciclo in cui si testa continuamente il valore di una variabile, il compilatore può effettuare delle ottimizzazioni sul codice, inserendo la variabile in un registro in memoria, in modo da rendere più veloce l'accesso.

Così facendo però non si potranno vedere le modifiche che occorrono, perché un dispositivo o un altro processo, accede all'indirizzo della variabile in memoria cambiandola.

Per evitare ciò basta dichiarare la variabile di tipo volatile

sig_chld.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

void sig_chld(int signum);
int main(int argc, char *argv[]);
volatile sig_atomic_t sons=10;

void sig_chld(int signum)
{int w;
 // finche' esiste un figlio che ha fornito il proprio status
 while(waitpid(WAIT_ANY,&w,WNOHANG)>0)
  if (WIFEXITED(w)) // se il figlio ha terminato
   sons--;// decrementa il numero di figli da attendere
}

int main(int argc, char *argv[])
{int nfork=10;
 int i;
 pid_t ok_fork;
 pid_t my_pid;
```

```
int w;
sigset_t block_sig;
sigset_t old_sig;

printf("Number of forks: %d\n",nfork);
my_pid=getpid();
signal(SIGCHLD,sig_chld); // modifica il gestore segnale
// genera i figli
for (i=0;i<nfork;i++)
 {if ((ok_fork=fork())<0)
  {printf("Error File %s, line %d\n",__FILE__,__LINE__);
   exit(EXIT_FAILURE);}
  // se sono un figlio
  if (!ok_fork)
  {// genera un intero casuale tra 1 e 10 da utilizzare come
   // tempo di attesa
   w=getpid();
   srand(w);
   w=1+rand()%10;
   printf("The %d-th child waits for %d sec\n",i,w);
   sleep(w); // attendi
   exit(EXIT_SUCCESS); // esci
   }
 }
```

```
 // inizializza la maschera di blocco dei segnali
 sigemptyset(&block_sig);
 sigaddset(&block_sig,SIGCHLD); // aggiungi SIGCHLD alla maschera
 // blocca SIGCHLD cosicche' se arriva tra la sospensione e la
 // lettura della variabile sons il processo non lo manchera'
 // restando in attesa perenne
 sigprocmask(SIG_BLOCK,&block_sig,&old_sig);
 // finche' ci sono figli superstiti
 while (sons)
 {// sblocca il segnale dalla maschera e contemporaneamente vai
  // in attesa di segnali. Implementato con un operazione atomica
  // di modo che se arriva un segnale tra la sospensione e la
  // lettura della variabile sons, il processo non lo manca e
  // quindi non resta in attesa perenne. Al ritorno sigsuspend
  // ripristina la maschera iniziale (cioe' con SIGCHLD bloccato)
  sigsuspend(&old_sig);
  printf("Current alive sons: %d...\n",sons);
 }
//sblocco finale di SIGCHLD per ripristinare la maschera iniziale
 sigprocmask(SIG_UNBLOCK,&block_sig,NULL);
 printf("All done\n"); exit(EXIT_SUCCESS); // esci
}
```

FIFO (o named pipe)

Le pipe possono essere utilizzate solo se due processi hanno un "antenato" comune, un processo crea la pipe e qualche discendente la usa.

Viceversa i file speciali FIFO possono essere utilizzate per consentire la comunicazione tra due processi arbitrari, essi devono condividere solo il "nome" della FIFO.
I file FIFO possono essere creati o attraverso il comando shell mkfifo, o attraverso la chiamata della funzione mkfifo.
Dopo la creazione della FIFO, essa può essere usata come "un file" utilizzando le funzioni open, read, write, close , ma non la lseek.

E' possibile che più processi scrivano sulla stessa FIFO e se il numero di byte scritti sulla FIFO è inferiore a PIPE_BUF, le scritture sono operazioni “atomiche”.

L'utilizzo di O_NONBLOCK consente di non bloccare le operazioni di open/read/write.

Come per le PIPE, se si esegue una write su di una FIFO che nessun processo ha aperto in lettura, si genera un signal SIGPIPE.

La sintassi della funzione mkfifo è la seguente:

```
int mkfifo(char *pathname, mode_t mode)
```

Con pathname percorso del file e mode specifiche di come aprire la fifo, del tutto identica a mode di open.

Se il file è aperto senza flag O_NONBLOCK allora:

- se la FIFO è aperta in sola lettura, la chiamata si blocca finché un altro processo non apre la FIFO in scrittura
- se la FIFO è aperta in sola scrittura, la chiamata si blocca finché un altro processo non apre la FIFO in lettura

Se il file è aperto con il flag O_NONBLOCK allora:

- se la FIFO è aperta in sola lettura, la chiamata ritorna immediatamente
- se la FIFO è aperta in sola scrittura, e nessun altro processo lo ha aperto in in lettura, la chiamata ritorna un codice di errore

Operazioni e Modalità

Operazione corrente	Status del descrittore complementare	Comportamento modalità bloccante	Comportamento modalità non bloccante
apertura FIFO in sola lettura	FIFO aperta in scrittura	ritorna con successo	ritorna con successo
	FIFO chiusa in scrittura	blocca finchè la FIFO è aperta in scrittura	ritorna con successo
apertuta FIFO in sola scrittura	FIFO aperta in lettura	ritorna con successo	ritorna con successo
	FIFO chiusa in lettura	blocca finchè la FIFO è aperta in lettura	ritorna con l'errore ENXIO
lettura da pipe o FIFO vuote	pipe o FIFO aperta in scrittura	blocca finchè sono immessi dati o il lato in scrittura viene chiuso	ritorna con l'errore EAGAIN
	pipe o FIFO chiusa in scrittura	ritorna col valore 0 (fine del file)	ritorna col valore 0 (fine del file)
scrittura su pipe o FIFO	pipe o FIFO aperta in lettura	Se #byte<= PIPE_BUF, scrive in modo atomico, bloccando se non c'è spazio disponibile. Se #byte>PIPE_BUF scrive in modo non atomico.	Se #byte<=PIPE_BUF, scrive in modo atomico, ritornando con EAGAIN se non c'è spazio disponibile. Se #byte>PIPE_BUF e non c'è almeno 1 byte disponibile, ritorna con EAGAIN; altrimenti scrive solo ciò che può.
	pipe o FIFO chiusa in lettura	genera SIGPIPE	genera SIGPIPE

Vediamo un esempio in cui un programma “Server” converte una stringa inviata da un programma “Client” in maiuscolo.

Fifoserver.c

```
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define MAX_BUF_SIZE 1000
int main(int argc, char *argv[])
{int fd, ret_val, count, numread;
 char buf[MAX_BUF_SIZE];
 /* Create the named - pipe */
 ret_val = mkfifo("miafifo", 0666);
 if ((ret_val == -1) && (errno != EEXIST))
  {perror("Error creating the named pipe");exit (1); }
    /* Open the pipe for reading */
    fd = open("miafifo", O_RDONLY);
    /* Read from the pipe */
   numread = read(fd, buf, MAX_BUF_SIZE);
```

```
    buf[numread] = '0';
    printf("Server : Read From the pipe : %s\n", buf);
    /* Convert to the string to upper case */
    count = 0;
    while (count < numread)
    {buf[count] = toupper(buf[count]);
     count++;
    }
 printf("Server : Converted String : %s\n", buf);
}
```

Fifoclient.c

```
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{int fd;
 /* Check if an argument was specified. */
 if (argc != 2)
 {printf("Usage:%s <string sent to the server>n",argv[0]);
```

```
  exit (1);
 }
/* Open the pipe for writing */
fd = open("miafifo", O_WRONLY);
/* Write to the pipe */
write(fd, argv[1], strlen(argv[1]));
}
```

Un altro esempio più completo che utilizza una fifo per comunicare dati tra due istanze dello stesso programma.

Fifo.c per eseguire: ./fifo [read | write]

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>
#define FIFO_NAME "fifo_tube"
#define PATH_MAX 1024
void to_fifo(char *file);
```

```c
void from_fifo(char *file);

int main(int argc, char *argv[]);
// legge dalla fifo
void from_fifo(char *file)
{int c;
 FILE *fstream;
 fstream=fopen(file,"r");// apre la fifo in lettura
 // finche' qualcuno ha aperto la fifo per scrivere leggi il
contenuto
 while ((c=fgetc(fstream))!=EOF) putchar(c);
 fclose(fstream); // chiudi la fifo
}
// scrive nella fifo
void to_fifo(char *file)
{int i;
 FILE *fstream;
 fstream=fopen(file,"w");// apri la fifo in scrittura
 fprintf(fstream,"something:\n"); // stampa header
 for (i=0;i<26;i++)// stampa dei caratteri random
   fputc('a'+(char)((rand()+getpid())%26),fstream);
 fprintf(fstream,"\n");
 fclose(fstream); // chiudi la fifo
 printf("Data sent\n");}
```

```
int main(int argc, char *argv[])
{int d=0;
 int fifo;
 char *cwd, cwd_[PATH_MAX];
 char *fifo_name=FIFO_NAME;
 char *full_fifo_name;
 int i, l;

 // se il programma non e' stato chiamato correttamente termina
 if (argc!=2)
 {printf("Error !!! File %s, line
%d\n",__FILE__,__LINE__);exit(EXIT_FAILURE);}

 // controlla se e' stato chiamato per leggere
 if (!strcmp("read",argv[1]))
    d=-1;
 // o scrivere nella fifo
 else if (!strcmp("write",argv[1]))
    d=1;
 // oppure esci
 else
  {printf("Error !!! File %s, line
```

```
%d\n",__FILE__,__LINE__);exit(EXIT_FAILURE);}
 // ottieni il percorso corrente
 cwd=getcwd(cwd_,PATH_MAX);
 // se il percorso e' maggiore del buffer allocato segnala
l'errore
 if ((!cwd) && (errno==ENAMETOOLONG))
  {printf("Error !!! File %s, line
%d\n",__FILE__,__LINE__);exit(EXIT_FAILURE);}
 /*
  * per evitare controlli e' possibile sostituire la riga di
codice
  * con:
  * cwd=get_current_dir_name();
  * e ricordarsi alla fine di liberare la memoria con
  * free(cwd);
  */
 // e genera il nome della fifo tenendo conto della lunghezza del
 // nome
 l=strlen(cwd)+2+strlen(fifo_name);
 // e dell'allocazione dello spazio necessario
 full_fifo_name=(char *)malloc(sizeof(char)*l);
 // copia la descrizione del percorso
 for (i=0;cwd[i]!='\0';i++)
```

```
    full_fifo_name[i]=cwd[i];
 // aggiungi il carattere "/"
 full_fifo_name[i++]='/';
 // e il nome da dare alla fifo
 for (l=0;fifo_name[l]!='\0';full_fifo_name[i++]=fifo_name[l++]);
 full_fifo_name[i]='\0';

 // prova a generare la fifo con i permessi di scrittura e
lettura
 if ((fifo=mkfifo(full_fifo_name,S_IRUSR|S_IWUSR)))
 {// se esiste continua (la prima instanza del programma e' gia'
  // in esecuzione)
  if (errno==EEXIST) printf("FIFO already created\n");
  else
  {printf("Error !!! File %s, line
%d\n",__FILE__,__LINE__);exit(EXIT_FAILURE);}
}

// avvisa se e' stata gia' creata la fifo
if (!fifo)
    printf("FIFO created\nWaiting...\n");

// leggi se devi
```

```
if (d==-1)
    from_fifo(full_fifo_name);

// scrivi altrimenti
if (d==1)
    to_fifo(full_fifo_name);

// elimina la fifo
remove(full_fifo_name);
// disalloca lo spazio utilizzato per memorizzare il nome della
fifo
free(full_fifo_name);

exit(EXIT_SUCCESS);
}
```