

Sincronizzazione tra processi in Unix: i segnali

Sincronizzazione tra processi

I processi interagenti possono avere bisogno di **sincronizzazione**.

Unix: non c'è condivisione di variabili tra processi:

➤ la sincronizzazione può realizzarsi mediante i segnali

Segnale:

è un'**interruzione software** a un processo, che notifica un evento asincrono; ad esempio segnali:

- generati da terminale (es. CTRL+C)
- generati da altri processi
- generati dal kernel in seguito ad eccezioni HW (violazione dei limiti di memoria, divisioni per 0, etc.)
- generati dal kernel in seguito a condizioni SW (time-out, scrittura su pipe chiusa, etc.)

Segnali Unix

- Un segnale può essere inviato:
 - dal kernel a un processo
 - da un processo utente ad altri processi utente(Es: comando **kill**)
- Quando un processo riceve un segnale, può comportarsi in **tre modi diversi**:
 1. **gestire** il segnale con una funzione *handler* definita dal programmatore
 2. **eseguire** un'azione predefinita dal S.O. (azione di *default*)
 3. **ignorare** il segnale (nessuna reazione)
- Nei primi due casi, il processo **reagisce in modo asincrono** al segnale:
 1. interruzione dell'esecuzione
 2. esecuzione dell'azione associata (*handler* o *default*)
 3. ritorno alla prossima istruzione del codice del processo interrotto

Segnali Unix

- Per ogni versione di Unix esistono vari tipi di segnale (in Linux, 32 segnali), ognuno identificato da un intero.
- Ogni segnale, è associato a un particolare evento e prevede una specifica **azione di default**.
- È possibile riferire i segnali con identificatori simbolici (**SIGxxx**):
SIGKILL, SIGSTOP, SIGUSR1, etc.
- L'associazione tra nome simbolico e intero corrispondente (che dipende dalla versione di Unix) è specificata nell'header file **<signal.h>**.

Segnali Unix (linux): signal.h

```
#define SIGHUP          1      /* Hangup (POSIX). Action: exit */
#define SIGINT         2      /* Interrupt (ANSI). Action: exit */
#define SIGQUIT        3      /* Quit (POSIX). Action: exit, core dump*/
#define SIGILL         4      /* Illegal instr.(ANSI).Action: exit,core dump */
...
#define SIGKILL        9      /* Kill, unblockable (POSIX). Action: exit*/
#define SIGUSR1        10     /* User-defined signal 1 (POSIX). Action: exit*/
#define SIGSEGV        11     /* Segm. violation (ANSI). Act: exit,core dump */
#define SIGUSR2        12     /* User-defined signal 2 (POSIX).Act: exit */
#define SIGPIPE        13     /* Broken pipe (POSIX).Act: exit */
#define SIGALRM        14     /* Alarm clock (POSIX). Act: exit */
#define SIGTERM        15     /* Termination (ANSI). Act:exit*/
...
#define SIGCHLD        17     /* Child status changed (POSIX).Act: ignore */
#define SIGCONT        18     /* Continue (POSIX).Act. ignore */
#define SIGSTOP        19     /* Stop, unblockable (POSIX). Act: stop */
...
```

Gestione dei segnali

- Quando un processo riceve un segnale, può gestirlo in 3 modi diversi:
 - gestire il segnale con una funzione *handler* definita dal programmatore
 - eseguire un'azione predefinita dal S.O. (azione di *default*)
 - *ignorare* il segnale

NB. Non tutti i segnali possono essere gestiti esplicitamente dai processi: **SIGKILL** e **SIGSTOP** non sono nè **intercettabili**, nè **ignorabili**.

➤ qualunque processo, alla ricezione di **SIGKILL** o **SIGSTOP** esegue sempre l'**azione di default**.

System call **signal**

Ogni processo può gestire esplicitamente un segnale utilizzando la system call **signal**:

```
void (* signal(int sig, void (*func)()))(int);
```

- **sig** è l'intero (o il nome simbolico) che individua il segnale da gestire
- il parametro **func** è un puntatore a una funzione che indica l'azione da associare al segnale; in particolare **func** può:
 - ✓ puntare alla routine di gestione dell'interruzione (*handler*)
 - ✓ valere **SIG_IGN** (nel caso di segnale ignorato)
 - ✓ valere **SIG_DFL** (nel caso di azione di default)
- ritorna un puntatore a funzione:
 - ✓ al precedente gestore del segnale
 - ✓ **SIG_ERR(-1)**, nel caso di errore

signal

Ad esempio:

```
#include <signal.h>
void gestore(int);
...
main()
{...
signal(SIGUSR1, gestore); /*SIGUSR1 gestito */
...
signal(SIGUSR1, SIG_DFL); /*USR1 torna a default */
signal(SIGKILL, SIG_IGN); /*errore! SIGKILL non è
                             ignorabile */
...
}
```

Routine di gestione del segnale (*handler*):

Caratteristiche:

- l'*handler* prevede sempre **un parametro formale** di tipo **int** che rappresenta il numero del segnale effettivamente ricevuto.
- l'*handler* non restituisce alcun risultato

```
void handler(int signum)
{ ....
  ....
  return;
}
```

Routine di gestione del segnale (*handler*):

Struttura del programma:

```
#include <signal.h>
void handler(int signum)
{ <istruzioni per la gestione del segnale>
  return;
}

main()
{ ...
  signal(SIGxxx, handler);
  ...
}
```

Gestione di segnali con handler

- Non sempre l'associazione *segnale/handler* è durevole:
 - alcune implementazioni di Unix (BSD, SystemV r.3 e seg.), prevedono che l'azione rimanga installata anche dopo la ricezione del segnale.
 - in alcune realizzazioni (SystemV, prime versioni), invece, dopo l'attivazione dell'handler ripristina automaticamente l'azione di default. In questi casi, per riagganciare il segnale all'handler:

```
main()  
{ ..  
  signal(SIGUSR1, f);  
  ...}
```

```
void f(int s)  
{ signal(SIGUSR1, f);  
  ....  
}
```

Esempio: parametro del gestore

```
/* file segnalil.c */
#include <signal.h>
void handler(int);

main()
{ if (signal(SIGUSR1, handler)==SIG_ERR)
    perror("prima signal non riuscita\n");
  if (signal(SIGUSR2, handler)==SIG_ERR)
    perror("seconda signal non riuscita\n");
  for (;;)
}

void handler (int signum)
{ if (signum==SIGUSR1) printf("ricevuto sigusr1\n");
  else if (signum==SIGUSR2) printf("ricevuto sigusr2\n");
}
```

Esempio: esecuzione & comando kill

```
anna@lab3-linux:~/esercizi$ vi segnal1.c
anna@lab3-linux:~/esercizi$ cc segnal1.c
anna@lab3-linux:~/esercizi$ a.out&
[1] 313
anna@lab3-linux:~/esercizi$ kill -SIGUSR1 313
anna@lab3-linux:~/esercizi$ ricevuto sigusr1

anna@lab3-linux:~/esercizi$ kill -SIGUSR2 313
anna@lab3-linux:~/esercizi$ ricevuto sigusr2

anna@lab3-linux:~/esercizi$ kill -9 313
anna@lab3-linux:~/esercizi$
[1]+  Killed                  a.out
anna@lab3-linux:~/esercizi$
```

Esempio: gestore del SIGCHLD

- **SIGCHLD** è il segnale che il kernel invia a un processo padre quando un figlio termina.
- È possibile svincolare il padre da un'attesa esplicita della terminazione del figlio, mediante un'apposita funzione *handler* per la gestione di **SIGCHLD**:
 - la funzione *handler* verrà attivata in modo **asincrono** alla ricezione del segnale
 - handler chiamerà la **wait** con cui il padre potrà raccogliere ed eventualmente gestire lo stato di terminazione del figlio

Esempio: gestore del SIGCHLD

```
#include <signal.h>
void handler(int);

main()
{ int PID, i;
  PID=fork();
  if (PID>0) /* padre */
  {   signal(SIGCHLD,handler);
      for (i=0; i<10000000; i++); /* attività del padre..*/
      exit(0); }
  else /* figlio */
  {   for (i=0; i<1000; i++); /* attività del figlio..*/
      exit(1); }
}

void handler (int signum)
{ int status;
  wait(&status);
  printf("stato figlio:%d\n", status>>8);}
```

Segnali & fork

Le associazioni segnali-azioni vengono registrate nella *User Area* del processo. Sappiamo che:

- una **fork** copia la *User Area* del padre nella *User Area* del figlio
- padre e figlio condividono lo stesso codice

quindi

- il figlio eredita dal padre le informazioni relative alla gestione dei segnali:
 - ignora gli stessi segnali ignorati dal padre
 - gestisce con le stesse funzioni gli stessi segnali gestiti dal padre
 - i segnali a default del figlio sono gli stessi del padre
- successive **signal** del figlio non hanno effetto sulla gestione dei segnali del padre.

Segnali & exec

Sappiamo che:

- una **exec** sostituisce codice e dati del processo che la chiama
- la *User Area* viene mantenuta, tranne le informazioni legate al codice del processo (ad esempio, le funzioni di gestione dei segnali, che dopo l'**exec** non sono più visibili!)

quindi

- dopo un'**exec**, un processo:

- ignora gli stessi segnali ignorati prima di **exec**
- i segnali a default rimangono a default

ma

- **i segnali che prima erano gestiti, vengono riportati a default**

Esempio

```
/* file segnali2.c */
#include <signal.h>

main()
{

    signal(SIGINT, SIG_IGN);
    execl("/bin/sleep", "sleep", "30", (char *)0);
}
```

**N.B. Il comando: `sleep N`
mette nello stato *sleeping* il processo per N secondi**

Esempio: esecuzione

```
anna@lab3-linux:~/esercizi$ cc segnali2.c
anna@lab3-linux:~/esercizi$ a.out&
[1] 500
anna@lab3-linux:~/esercizi$ kill -SIGINT 500
anna@lab3-linux:~/esercizi$ kill -9 500
anna@lab3-linux:~/esercizi$
[1]+  Killed                  a.out
anna@lab3-linux:~/esercizi$
```

System call `kill`

I processi possono inviare segnali ad altri processi con la `kill`:

```
int kill(int pid, int sig);
```

- **sig** è l'intero (o il nome simb.) che individua il segnale da gestire
- il parametro **pid** specifica il destinatario del segnale:
 - ✓ `pid > 0`: l'intero è il pid dell'unico processo destinatario
 - ✓ `pid = 0`: il segnale è spedito a tutti i processi appartenenti al gruppo del mittente
 - ✓ `pid < -1`: il segnale è spedito a tutti i processi con `groupId` uguale al valore assoluto di `pid`
 - ✓ `pid == -1`: vari comportamenti possibili (Posix non specifica)

kill: esempio:

```
#include <stdio.h>
#include <signal.h>
int cont=0;
void handler(int signo)
{ printf ("Proc. %d: ricevuti n. %d segnali %d\n",
  getpid(),cont++, signo);
}

main ()
{int pid;
  pid = fork();
  if (pid == 0) /* figlio */
  { signal(SIGUSR1, handler);
    for (;;)
    }
  else /* padre */
    for(;;) kill(pid, SIGUSR1);
}
```

Segnali: altre system call

sleep:

```
unsigned int sleep(unsigned int N)
```

- provoca la sospensione del processo per N secondi (al massimo)
- se il processo riceve un segnale durante il periodo di sospensione, viene risvegliato *prematuramente*
- ritorna:
 - ✓ 0, se la sospensione non è stata interrotta da segnali
 - ✓ se il risveglio è stato causato da un segnale al tempo N_s , **sleep** restituisce in numero di secondi non utilizzati dell'intervallo di sospensione ($N-N_s$).

Segnali: altre system call

alarm:

```
unsigned int alarm(unsigned int N)
```

- imposta in timer che dopo N secondi invierà al processo il segnale **SIGALRM**
- ritorna:
 - ✓ 0, se non vi erano time-out impostati in precedenza
 - ✓ il numero di secondi mancante allo scadere del time-out precedente

NB: l'azione di *default* associata a SIGALRM è la terminazione.

Segnali: altre system call

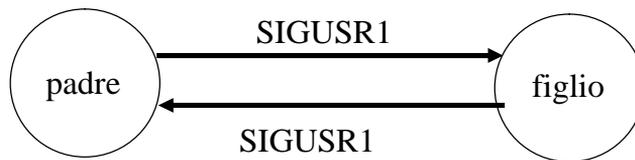
pause:

```
int pause(void)
```

- sospende il processo fino alla ricezione di un qualunque segnale
- ritorna -1 (`errno = EINTR`)

Esempio

Due processi (padre e figlio) che si sincronizzano alternativamente mediante il segnale SIGUSR1 (gestito da entrambi con la funzione *handler*):



```
int ntimes = 0;
void handler(int signo)
{printf ("Processo %d ricevuto #%d volte il segnale %d\n",
    getpid(), ++ntimes, signo);
}
```

```

main ()
{int pid, ppid;
  signal(SIGUSR1, handler);
  if ((pid = fork()) < 0) /* fork fallita */
    exit(1);
  else if (pid == 0) /* figlio*/
  {  ppid = getppid(); /* PID del padre */
    for (;;)
    {  printf("FIGLIO %d\n", getpid());
      sleep(1);
      kill(ppid, SIGUSR1);
      pause();}
    }
  else /* padre */
    for(;;) /* ciclo infinito */
    {  printf("PADRE %d\n", getpid());
      pause();
      sleep(1);
      kill(pid, SIGUSR1); }}

```

Affidabilità dei segnali

Aspetti:

- il gestore rimane **installato**?
 - Se no: posso reinstallare all'interno dell'handler

```
void handler(int s)
{ signal(SIGUSR1, handler);
  printf("Processo %d: segnale %d\n", getpid(), s);
  ... }
```

Cosa succede se qui
arriva un nuovo
segnale?

- cosa succede se arriva il segnale durante l'esecuzione dell'handler?
 - **innestamento** delle routine di gestione
 - **perdita** del segnale
 - **accodamento** dei segnali (segnali *reliable*, *BSD 4.2*)

Interrompibilità di System Calls

- System Call: possono essere classificate in
 - *slow* system call: possono richiedere tempi di esecuzione non trascurabili perchè possono causare periodi di attesa (es: lettura da un dispositivo di I/O lento)
 - system call *non slow*.
- una *slow* system call è interrompibile da un segnale; in caso di interruzione:
 - ✓ ritorna -1
 - ✓ **errno** vale EINTR
- possibilità di ri-esecuzione della system call:
 - ✓ automatica (BSD 4.3)
 - ✓ non automatica, ma comandata dal processo (in base al valore di **errno** e al valore restituito)