SINCRONIZZAZIONE IN UNIX

SINCRONIZZAZIONE ===> SEGNALI

In UNIX abbiamo un insieme di segnali
===> simili al verificarsi di un **interrupt hardware**

Il processo esegue l'azione di trattamento del segnale come un ***unico thread di esecuzione***

Segnale:
è un'**interruzione 'virtualizzata' e inviata tramite il kernel** a un processo, che notifica un *evento* (asincrono/sincrono)

Un segnale può essere inviato:
1) dal kernel a un processo
2) da un processo utente ad altri processi utente

Esempi di segnali:

1)
- generati da terminale (es. **CTRL+C**) ← evento asincrono!
- generati da eccezioni HW (violazione dei limiti di memoria, etc.) ← evento sincrono!
- generati da condizioni SW (divisioni per 0, scrittura su pipe chiusa, etc.) ← evento sincrono!

2) generati da altri processi ← evento asincrono!

Se un processo termina a causa di un segnale, il valore di **status** ritornato al padre che attende con una **wait** è:
8 bit bassi ⇨ numero del segnale
8 bit alti ⇨ 0 (zero)

UNIMORE

SEGNALI (segue)

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, rispetto alla esecuzione del processo che riceve il segnale, **reagisce** nel seguente modo:

a. interruzione dell'esecuzione del programma che il processo sta eseguendo
b. esecuzione dell'azione associata (***handler*** o ***default***)
c. ritorno alla istruzione successiva del codice del programma interrotto

I segnali quindi rappresentano la notifica del verificarsi di **eventi** che il processo riceve e può nei casi 1. e 2. trattare con:

* ***gestori/handler specifici*** del segnale
* ***azioni a default***
  ===> in genere, ***TERMINAZIONE***

OSSERVAZIONI:

I segnale **non sono persistenti**
Quindi se un segnale non può essere ricevuto da alcun processo ⇨ **perso**

Viceversa se più processi possono ricevere lo stesso segnali ⇨ **tutti avvertiti**

ELENCO DEI SEGNALI

(contenuto nel file **signal.h**)

nome	***numero***	
SIGHUP	1	hangup: sconnessione del terminale (logout)
SIGINT	2	interrupt da terminale (in genere, **<CTRL>C**)
SIGQUIT	3	quit da un programma con core (**<CTRL>**)
SIGILL	4	istruzione non consentita (core)
...		
SIGKILL	9	uccisione (*non intercettabile o ignorabile*)
...		
SIGSYS	12	errore di argomento nella system call (core)
SIGPIPE	13	scrittura su pipe che non ha lettore
SIGALRM	14	allarme da orologio
SIGTERM	15	terminazione software
SIGUSR1	16	user interrupt 1
SIGUSR2	17	user interrupt 2
SIGCHLD	18	morte di un child
...		

segnali lasciati all'utente

⇨ *default*: **NO TERMINAZIONE**

ESEMPI DI USO DI SEGNALI:

1) # prog
programma che è entrato in un loop infinito
<CTRL>-C
⇨ spedizione del segnale **SIGINT** al processo
⇨ azione di DEFAULT ===> TERMINAZIONE

2) # prog
Illegal instruction - core dumped
programma (corrotto) che ha eseguito una istruzione illegale
⇨ spedizione del segnale **SIGILL** al processo
⇨ azione di DEFAULT: TERMINAZIONE con produzione di un file **core** (immagine del processo)

3) # prog &
<PID>
...
kill <PID>
<PID> Terminated
⇨ spedizione del segnale **SIGTERM** al processo
⇨ azione di DEFAULT: TERMINAZIONE

4) # prog &
<PID>
...
kill **-9** <PID>
<PID> Terminated
⇨ spedizione del segnale **SIGKILL** al processo
questo segnale non può essere IGNORATO nè associato ad un gestore
⇨ azione di DEFAULT: TERMINAZIONE

PRIMITIVE SIGNAL
===> DEFINIZIONE di HANDLER

SIGNAL: #include <signal.h>
void (***signal**(int sig, void (*func)(int))) (int);
/* func è un puntatore a funzione */

Si specifica
- ***quale*** segnale (sig) (**NON** può essere SIGKILL)
- ***come*** trattare il segnale (func)

TRE POSSIBILITÀ (da slide 2)**:**

1) specificare l'indirizzo di una funzione **GESTORE/HANDLER** del segnale
```
void f(int sig) {...} /* definizione GESTORE */
signal(sig,f);
```
all'occorrenza del segnale **sig** viene invocata **f**
che riceve il numero del segnale come argomento (sig)
⇨ **f** può o semplicemente eseguire certe azioni e quindi poi il processo **continuerà** nell'esecuzione del programma interrotto oppure può **terminare** l'esecuzione del processo (se prevede una exit nel suo codice)

2) riportare all'azione di default (SIG_DFL)
===> terminazione del processo (in genere)
```
signal(sig,SIG_DFL);
```
===> riporta all'azione di **default**

3) **ignorare** il segnale (SIG_IGN)
```
signal(sig,SIG_IGN);
```
===> sig viene **ignorato** dal processo

OSSERVAZIONI sui SEGNALI:

1) I **SEGNALI** hanno tutti la stessa **priorità**

2) L'azione, dopo che il segnale si è presentato, può:

   * essere riportata al valore di default
     ⇨ **comportamento di UNIX SystemV**

   * rimanere installata
     ⇨ **comportamento di UNIX BSD**

NOTA BENE: bisogna verificare nella propria versione di UNIX/LINUX quale 'filosofia' si segue!

3) Nel caso di uso di una delle primitive della famiglia EXEC: i segnali ignorati e collegati alle azioni di default rimangono tali, mentre quelli agganciati a gestori/handler specifici vengono riportati all'azione di default ➔ perché?

4) Nello schema di processo figlio (generato da un processo SHELL) **nel caso di esecuzione in background**, il processo prima di eseguire la EXEC procede ad ignorare i segnali SIGINT (**<CTRL>C**) e SIGQUIT ➔ perché?

Segnali & fork

Le associazioni segnali-azioni vengono registrate nella *Kernel Area* del processo

Sappiamo che:

- una **fork** copia il contenuto della ***Kernel Area*** del padre nella ***Kernel Area*** del figlio
- padre e figlio condividono lo stesso codice

quindi

- il figlio eredita dal padre le informazioni relative alla gestione dei segnali:
  - gestisce con le stesse funzioni handler gli stessi segnali gestiti dal padre
  - ignora gli stessi segnali ignorati dal padre
  - i segnali collegati al default del figlio sono gli stessi del padre

☞ Chiaramente le successive **signal** del figlio non hanno effetto sulla gestione dei segnali del padre e viceversa

Segnali & exec

Sappiamo che:

- una **exec** sostituisce codice e dati del processo che la chiama
- la ***Kernel Area*** viene mantenuta, mentre le informazioni legate al codice del processo precedente sono perse

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

- ignora gli stessi segnali ignorati prima di **exec**
- i segnali collegati al default rimangono collegati al default
- **i segnali che prima erano gestiti, vengono riportati a default** (dato che le funzioni di gestione dei segnali dopo l'**exec** non sono più visibili!)

ESEMPIO di trattamento di segnale:

Si intercetta il segnale **SIGINT** collegato al **<CTRL>C**: il programma rimane in un **ciclo senza fine**, con incremento di un contatore. L'invio del segnale produce una semplice stampa, poi si torna ad eseguire il programma. Come si esce? Tentare altri segnali non intercettati

COMPORTAMENTO UNIX BSD

```
#include <signal.h>

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

main ()
{ int i;

   signal(SIGINT, catchint);
       /* si aggancia il segnale */
   for (;;) /* ciclo infinito */
      for (i =0 ; i < 32000; i++)
             printf (" i vale %d\n", i);
}
```

ESEMPIO di trattamento di segnale (segue):

COMPORTAMENTO UNIX System V

```
#include <signal.h>

void catchint(int signo)
{                    /*  ☜  POSSIBILI  PROBLEMI
*/
   signal(SIGINT, SIG_IGN);
   /* si disabilita il segnale SIGINT */

   printf ("\n catchint: signo=%d\n", signo);
   printf ("ciao \n");
  /* non si prevedono azioni di terminazione:
  ritorno    al    segnalato,    dopo    aver
  ripristinato la catch function */
   signal(SIGINT, catchint);
}

main ()
{  int i;

   signal(SIGINT, catchint);
       /* si aggancia il segnale */

   for (;;)   /* ciclo infinito */
      for (i =0 ; i < 32000; i++)
             printf (" i vale %d\n", i);
}
```

INVIO DI SEGNALI

DA UN PROCESSO AD ALTRI PROCESSI

I processi possono inviare segnali ad altri processi con la primitiva

KILL retval = **kill**(pid, sig);
int retval;
int pid, sig;

- il parametro **`pid`** specifica il processo destinatario del segnale
- **`sig`** è l'intero (o il nome simbolico-macro) che individua il segnale da gestire

Questa primitiva INVIA il segnale **sig** al processo con **pid** specificato

NOTA:
L'identificatore effettivo dell'utente che effettua la kill DEVE essere uguale a quello del processo a cui si invia il segnale

CASI PARTICOLARI:

- ✓ pid == 1: è il pid del processo INIT
- ✓ sig == 0: si verifica se il PID è valido

ALTRE PRIMITIVE:

1) Sospensione in attesa di un qualunque segnale

int **pause** ();

Sospende il processo fino alla ricezione di un qualunque segnale

2) Sospensione temporizzata

unsigned int **sleep** (numerosecondi);
unsigned int numerosecondi;

Provoca la sospensione del processo per N secondi: 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 Ns, **sleep** restituisce in numero di secondi non utilizzati dell'intervallo di sospensione (N-Ns)

3) Installazione di un allarme

unsigned int **alarm** (numerosecondi);
unsigned int numerosecondi;

Imposta il 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.

ESEMPIO: Uso di kill e pause

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

```
#define BELL= "\007\007\007\007"
#include <signal.h>

int ntimes = 0;
/* variabile contatore globale per le funzioni C:
due copie una per ogni processo */

void handler(int signo)
{ printf("Processo %d ricevuto #%d volte il segnale
%d\n", getpid(), ++ntimes, signo);
   printf(BELL);
   signal(SIGUSR1, handler);
   /* necessaria SOLO per UNIX System V */
}

main ()
{ int pid, ppid;

/* aggancia inizialmente il segnale per il padre:
   si utilizza un segnale libero: SIGUSR1 */
   signal(SIGUSR1, handler);

   if ((pid = fork()) < 0) {exit (1); }
   else if (pid == 0) /* figlio */
     { /* l'installazione della catch function viene
     ereditata */
        ppid= getppid();   /* PID del padre */
    for (;;) { /* ciclo infinito */
          printf("FIGLIO %d\n", getpid());
          sleep(1);
          kill(ppid, SIGUSR1); /* invio all'altro */
          pause(); /* attesa del segnale */
 }
     }
```

```
   else /* padre */
     { for (;;) { /* ciclo infinito */
           printf (" PADRE %d\n", getpid());
           pause(); /* attesa iniziale */
           sleep(1);
           kill(pid, SIGUSR1); /* invio segnale */
 } }
}
```

Mediante questo schema i processi padre e figlio si passano alternatamente il controllo

Padre

pause();
sleep(1);
kill(...);

Figlio

sleep(1);
kill(...);
pause();

ESEMPIO: Uso di alarm e pause

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

#define TRUE 1
#define FALSE 0
#define BELLS "\007\007\007"

int alarm_flag = FALSE;
/* variabile globale per le funzioni C: due copie
una per ogni processo */

/* catch function per il SEGNALE di ALLARME */
void setflag()
{  alarm_flag = TRUE;
   /* quando si verifica l'allarme si setta questa
variabile */
}

main (int argc, char *argv[])
{  int pid, nsecs, j;

if (argc <= 2)
   {  printf ("Errore nel numero di parametri\n");
      exit(1);
   }

if ((nsecs = atoi(argv[1]) * 60 ) <= 0)
   { printf ("Errore nel valore del tempo\n");
    exit(2);
   }
```

ESEMPIO: Uso di alarm e pause (segue)

```
switch (pid = fork())
{
   case -1: /* fork fallita */
       printf("Errore in fork\n");
       exit(1);
   case 0: /* figlio */
       break;
   default: /* padre */
       printf("Creazione del processo %d\n", pid);
       exit(0);
/* il padre realizza una esecuzione in background
del figlio */
}

/* figlio */
signal(SIGALRM, setflag);

alarm(nsecs);

pause();

if (alarm_flag == TRUE)
   { printf(BELLS);
      for (j=2; j < argc; j++)
         printf("%s ", argv[j]);
      printf("\n");
   }
exit (0);
}
```