POSIX - Gestione dei Segnali

E.Mumolo, DEEI

`mumolo@units.it`

Segnali in POSIX

- **Segnali in Posix**
  - Modalità di notifiche relative a vari eventi asincroni
  - I signal interrompono un processo e possono o meno essere gestiti
  - La tempistica di servizio non e' predicibile
  - I segnali sono dei numeri e dei nomi simbolici (kill -l elenca i segnali)

- **Uso dei segnali**
  - Gestione di eventi se catturati
  - Terminazione della esecuzione se non catturati

- **Invio dei segnali da tastiera**
  - ctrl-C → INT; ctrl-Z → TSTP; crtl-\ → ABRT
- **Invio dei segnali da linea di comando**
  - Kill -signal PID; fg : manda un segnale CONT
- **Invio dei segnali mediante chiamate di sistema**
  - Kill(); signal()

Segnali in POSIX

- **In POSIX :**
  - Capacità di bloccare le system call: standard
  - I signal handler rimangono installati: standard
  - Restart automatico delle system call: non specificato
    - In realtà, in molti S.O. moderni è possibile specificare se si desidera il restart automatico oppure no
- **POSIX specifica un meccanismo per segnali affidabili:**
  - E' possibile gestire ogni singolo dettaglio del meccanismo dei segnali
    - quali bloccare
    - quali gestire
    - come evitare di perderli, etc.

Segnali - generazione

- **Eccezioni hardware**
  - Divisione per 0 (**SIGFPE**)
  - Riferimento non valido a memoria (**SIGSEGV**)
  - L'interrupt viene generato dall'hardware, e catturato dal kernel; questi invia il segnale al processo in esecuzione
- **System call kill**
  - Permette di spedire un segnale ad un altro processo
  - Limitazione: uid del processo che esegue **kill** deve essere lo stesso del processo a cui si spedisce il segnale, oppure deve essere 0 (root)
- **Condizioni software**
  - Eventi asincroni generati dal software del sistema operativo, non dall'hardware della macchina. Esempi:
    - terminazione di un child (**SIGCHLD**)
    - generazione di un alarm (**SIGALRM**)

Segnali - gestione

- **Quando un segnale viene ricevuto, si può:**
  - **Ignorare il segnale**
    - Alcuni segnali che non possono essere ignorati: **SIGKILL** e **SIGSTOP**
    - Motivo: permettere al superutente di terminare processi
    - Segnali hardware: comportamento non definito in POSIX se ignorati
  - **Eseguire l'azione di default**
    - Per molti segnali "critici", l'azione di default consiste nel terminare il processo
    - Può essere generato un file di core, tranne che nei seg. casi:
      - bit set-user-id e set-group-id settati e uid/gid diversi da owner/group;
      - mancanza di permessi in scrittura per la directory;
      - core file troppo grande
  - **Catturare ("catch") il segnale:**
    - Il kernel informa il processo chiamando una funzione specificata dal processo stesso (signal handler)
    - Il signal handler gestisce il problema nel modo più opportuno. **Esempio:**
      - nel caso del segnale **SIGCHLD** (terminazione di un child) → possibile azione: eseguire **waitpid**
      - nel caso del segnale **SIGTERM** (terminazione standard) → possibili azioni: rimuovere file temporanei, salvare file

Alcuni segnali importanti

SIGABRT (Terminazione, core)
• Generato da system call **abort()**; terminazione anormale
SIGALRM (Terminazione)
• Generato da un timer settato con la system call **alarm** o la funzione **setitimer**
SIGCHLD (Default: ignore)
• Quando un processo termina, **SIGCHLD** viene spedito al processo parent
• Il processo parent deve definire un signal handler che chiami **wait** o **waitpid**
SIGFPE (Terminazione, core)
• Eccezione aritmetica, come divisioni per 0
SIGHUP (Terminazione)
• Inviato ad un processo se il terminale viene disconnesso
SIGILL (Terminazione, core)
• Generato quando un processo ha eseguito un'azione illegale
SIGINT (Terminazione)
• Generato quando un processo riceve un carattere di interruzione (**Ctrl-C**) dal Terminale
SIGKILL (Terminazione)
• Maniera sicura per uccidere un processo
SIGPIPE (Terminazione)
• Scrittura su pipe/socket in cui il lettore ha terminato/chiuso
SIGSEGV (Terminazione, core)
• Generato quando un processo esegue un riferimento di memoria non valido
SIGTERM (Terminazione)
• Segnale di terminazione normalmente generato dal comando kill

Segnali - funzioni

- **La funzione sigaction() definisce la gestione dei segnali**

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

  - **signo**: l'identificatore del segnale che si vuole catturare
  - **act**:punta ad una struttura dove il programmatore specifica la nuova azione
  - **oact**: punta ad una struttura dove la funzione scrive l'ultima azione

Esempio di utilizzo:

```
int main() {
   int i;
   struct sigaction act, oact;

   act.sa_handler = funzione;
   If (sigaction(SIGALARM, &act, &oact !=0) perror("sigaction");

...
```

Segnali – kill system call

```
int kill(pid_t pid, int signo);
```

- **kill** spedisce un segnale ad un processo oppure a un gruppo di processi
- Argomento **pid**:
  - **pid > 0** spedito al processo identificato da pid
  - **pid == 0** spedito a tutti i processi appartenenti allo stesso gruppo del processo che invoca kill
  - **pid < -1** spedito al gruppo di processi identificati da –pid
  - **pid == -1** non definito
- Argomento **signo**:
  - Numero di segnale spedito

Segnali – kill system call (cont.)

```
int kill(pid_t pid, int signo);
```

- Permessi:
  - Il superutente può spedire segnali a chiunque
  - Altrimenti, il real uid o l'effective uid della sorgente deve essere uguale al real uid o l'effective uid della destinazione
- POSIX.1 definisce il segnale 0 come il *null signal*
- Se il segnale spedito è null, **kill** esegue i normali meccanismi di controllo errore senza spedire segnali
  - Esempio: verifica dell'esistenza di un processo; spedizione del null signal al processo (nota: i process id vengono riciclati)

```
int raise(int signo);
```

- Questa system call spedisce il segnale al processo chiamante

Segnali

```
unsigned int alarm(unsigned int sec);
```

- Questa SC permette di creare un allarme che verrà generato dopo il numero specificato di secondi
- Allo scadere del tempo, il segnale **SIGALRM** viene generato
- Attenzione: *il sistema non è real-time*
  - Garantisce che la pausa sarà almeno di **sec** secondi
  - Il meccanismo di scheduling può ritardare l'esecuzione di un processo
- Esiste un unico allarme per processo
  - Se un allarme è già settato, il numero di secondi rimasti prima dello scadere viene ritornato da alarm
  - Se **sec** è uguale a zero, l'allarme preesistente viene generato

Segnali

```
unsigned int alarm(unsigned int sec);
```

- L'azione di default per **SIGALRM** è di terminare il processo
- Ma normalmente viene definito un signal handler per il segnale

```
int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *value, struct itimerval
   *ovalue);
```

- Queste SC permettono un controllo più completo

```
int pause();
```

- Questa SC sospende il processo fino a quando un segnale non viene catturato
- Ritorna –1 e setta **errno** a **EINTR**

```
// questo programma cattura i segnali definiti dall'utente e stampa un messaggio di errore
#include  <signal.h>
#include  <unistd.h>

static void sig_usr(int);

Int main(void)
{
  if (signal(SIGUSR1, sig_usr) == SIG_ERR)
    perror("can't catch SIGUSR1");
  if (signal(SIGUSR2, sig_usr) == SIG_ERR)
    perror("can't catch SIGUSR2");

  for ( ; ; )
    pause();
}

static void sig_usr(int signo)    /* l'argomento e' un numero di segnale */
{
  if (signo == SIGUSR1)
    printf("received SIGUSR1\n");
  else if (signo == SIGUSR2)
    printf("received SIGUSR2\n");
  else {
    printf("received signal %d\n", signo);
    abort();
    exit(1);
  }
  return;
}
```

Segnali

```
unsigned int sleep(unsigned int seconds);
```

- questa system call causa la sospensione del processo fin a quando:
  - Non trascorre l'ammontare di tempo specificato
    - In questo caso ritorna 0
  - un segnale viene catturato e il signal handler effettua un return
    - In questo caso ritorna il tempo rimasto prima del completamento della sleep
- nota:
  - la sleep può concludersi dopo il tempo richiesto
  - la sleep può essere implementata utilizzando alarm(), ma spesso questo non accade per evitare conflitti

Segnali

`void abort();`

- questa system call spedisce il segnale **SIGABRT** al processo
- comportamento in caso di:
  - **SIG_DFL**: terminazione del processo
  - **SIG_IGN**: non ammesso
  - signal handler: il segnale viene catturato
- nel caso in cui il segnale viene catturato, il signal handler:
  - può eseguire return
  - può invocare **exit** o **_exit**
- in entrambi i casi, il processo viene terminato
- motivazioni per il catching: cleanup

Segnali

- **Quando un programma esegue una system call fork:**
  - I signal catcher settati nel parent vengono copiati nel figlio
- **Quando un programma viene eseguito tramite exec**
  - Se il signal catcher per un certo segnale è default o ignore, viene lasciato inalterato nel child
  - Se il signal catcher è settato ad una particolare funzione, viene cambiato a default nel child
    - Motivazione: la funzione selezionata può non esistere più nel figlio
- **Casi particolari**
  - Quando un processo viene eseguito in background
    - Segnali **SIGINT** e **SIGQUIT** vengono settati a ignore
    - In che momento / da chi questa operazione viene effettuata?