Modulo 7
System call relative ai segnali

Laboratorio di Sistemi Operativi I
Anno Accademico 2008-2009

Introduzione ai segnali - I

- **I segnali sono interruzioni software a livello di processo**
  - Permettono la gestione di eventi asincroni che interrompono il normale funzionamento di un processo
  - La gestione avviene tramite *signal handler*
  - POSIX.1 standardizza la gestione dei segnali

Introduzione ai segnali - II

- **Caratteristiche dei segnali**
  - Ogni segnale ha un identificatore
    - Identificatori di segnali iniziano con i tre caratteri SIG
    - Es. **`SIGABRT`** è il segnale di abort
  - Numero segnali: 15-40, a seconda della versione di UNIX
    - POSIX: 18
    - Linux: 38
  - I nomi simbolici corrispondono ad un intero positivo
    - Definizioni di costanti in **`bits/signum.h`**
    - Il numero 0 è utilizzato per un caso particolare

Condizioni che possono generare segnali - I

- **Pressione di tasti speciali sul terminale**
  - Es: Premere il tasto **`ctrl-c`** genera il segnale **`SIGINT`**
- **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 0 (root)

Condizioni che possono generare segnali - II

- **Comando `kill`**
  - Interfaccia shell alla system call **`kill`**
- **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`**)

Azioni associate ai segnali - I

- **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
- **Esecuzione dell'azione di default**
  - Per molti segnali "critici", l'azione di default consiste nel terminare il processo
  - Può essere generato un file di core (eccetto quando bit set-user-id e set-group-id settati e uid/gid sono diversi da owner/group o mancano di permessi in scrittura per la directory il core file e' troppo grande)

Azioni associate ai segnali - II

- **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 dei segnali più importanti - I

- **`SIGABRT` (Terminazione, core)**
  - Generato da syscall `abort()`; terminazione anormale
- **`SIGALRM` (Terminazione)**
  - Generato da un timer settato con la syscall `alarm()` o la funzione `setitimer()`
- **`SIGBUS` (Non POSIX; terminazione, core)**
  - Indica un hardware fault (definito dal s.o.)
- **`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

Alcuni dei segnali più importanti - II

- **`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
- **`SIGIO` (Non POSIX; default: terminazione, ignore)**
  - Evento I/O asincrono
- **`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

Alcuni dei segnali più importanti - III

- **`SIGUSR1`, `SIGUSR2` (Terminazione)**
  - Segnali non definiti utilizzabili a livello utente
- **`SIGSTP` (Default: stop process)**
  - Generato quando un processo riceve un carattere di suspend (**`ctrl-z`**) dal terminale
- **`SIGSYS` (Terminazione, core)**
  - Invocazione non valida di system call
  - Esempio: parametro errato
- **`SIGTERM` (Terminazione)**
  - Segnale di terminazione normalmente generato dal comando **`kill`**
- **`SIGURG` (Non POSIX; ignora)**
  - Segnala il processo che una condizione urgente è avvenuta (dati out-of-bound ricevuti da una connessione di rete)

System call signal - I

- `void (*signal(int signo, void (*func)(int)))(int);`
- **Descrizione:**
  - **`signo`**: l'identificatore del segnale che si vuole catturare
  - **`func`**: l'azione che vogliamo che sia eseguita
    - **`SIG_IGN`**: ignora il segnale (non applicabile a **`SIGKILL`** e **`SIGSTOP`**)
    - **`SIG_DFL`**: azione di default
    - l'indirizzo del signal handler: quando si vuole catturare il segnale (non applicabile a **`SIGKILL`** e **`SIGSTOP`**)
  - valore di ritorno:
    - il valore del precedente signal handler se ok
    - SIG_ERR in caso di errore

System call signal - II

- **Definizione alternativa (più chiara):**

```
typedef void sighandler_t(int);

sighandler_t *signal(int, sighandler_t*);
```

- **Definizione delle costanti (tipica) in `signal.h`:**
  - Queste costanti possono essere utilizzate come "puntatori a funzioni che prendono un intero e non ritornano nulla"
  - I valori devono essere tali che non possano essere assegnati a signal handler

```
#define SIG_ERR (void (*)())-1

#define SIG_DFL (void (*)())0

#define SIG_IGN (void (*)())1
```

Esercizio 1

- **Scrivere un programma che:**
  - Cattura i segnali definiti dall'utente (SIGUSR1, SIGUSR2) e stampa un messaggio di ricezione
- **Esempio di output:**

```
$ a.out &
[1]       235
$ kill –USR1 235      # spedisci segnale SIGUSR1 a PID 235
received SIGUSR1      # catturato
$ kill 235            # spedisci segnale SIGTERM
[1] + Terminated      a.out &
```

Generazione dei segnali - I

- **System call:** `int kill(pid_t pid, int signo);`
  - La system call **`kill`** spedisce un segnale ad un processo oppure ad 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

Generazione dei segnali - II

- **System call:** `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 (N.B: i process id vengono riusati!)
- **System call:** `int raise(int signo);`
  - Spedisce il segnale al processo chiamante

Generazione dei segnali - III

- **System call:** `unsigned int alarm(unsigned int sec);`
  - Questa system call 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

Generazione dei segnali - IV

- **System call:** `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
- **System call:**

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

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

  - Permettono un controllo più completo
- **System call:** `int pause();`
  - Questa syscall sospende il processo fino a quando un segnale non viene catturato (ritorna –1 e setta **`errno`** a **`EINTR`**)

Esercizio 2 - I

- **Scrivere un programma che:**
  - Genera NUMCHLD processi figli, ognuno dei quali dovra' terminare con un *exit status* diverso
  - Intercetta SIGCHLD per ogni processo figlio che termina e stampa un messaggio di avviso
  - Alla terminazione di tutti i figli, stampa un messaggio di avviso seguito dalla lista dei loro *PID* ed *exit status* ed infine esce
- **Attenzione a quale funzione (main, signal handler o altro) deve svolgere ciascuna delle operazioni definite sopra!**

Esercizio 2 - II

- Esempio di output:

```
[francesco@fpedulla signalExamples]$ ./a.out

Figlio [2349] generato correttamente...

Ricevuto segnale SIGCHLD!

Figlio [2350] generato correttamente...

Ricevuto segnale SIGCHLD!

Figlio [2351] generato correttamente...

Ricevuto segnale SIGCHLD!

Ricevuti tutti i segnali SIGCHLD!

Figlio [2349] terminato con stato 0.

Figlio [2350] terminato con stato 256.

Figlio [2351] terminato con stato 512.
```

Funzione sleep

- `unsigned int sleep(unsigned int seconds);`
  - questa system call causa la sospensione del processo fino a quando:
    - l'ammontare di tempo specificato trascorre
      - return value: 0
    - un segnale viene catturato e il signal handler effettua un return
      - return value: 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

Funzione abort

- `void abort();`
  - questa funzione spedisce il segnale **`SIGABRT`** al processo
  - comportamento in caso di:
    - **`SIG_DFL`**: terminazione del processo
    - **`SIG_IGN`**: non ammesso
    - signal handler installato: il segnale viene catturato e, prima che il processo venga terminato,
      - il signal handler può eseguire return
      - il signal handler può invocare **`exit`** o **`_exit`**
  - motivazioni per il catching: cleanup

Startup

- **Quando un programma esegue una system call `fork()`:**
  - I signal handler definiti nel parent vengono copiati nel figlio
- **Quando un programma viene eseguito tramite `exec()`**
  - Se il signal handler per un certo segnale è default o ignore, viene lasciato inalterato nel child
  - Se il signal handler è settato ad una particolare funzione, viene cambiato a **`default`** nel child
    - Motivazione: la funzione può non esistere 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?

Segnali e funzioni *rientranti*

- L'utilizzo dei segnali pone un nuovo problema relativo alla corretta esecuzione del codice di un programma, poiche':
  - la normale sequenza di istruzioni **viene interrotta**, per saltare all'esecuzione delle istruzioni del *signal handler,* e la coerenza delle strutture dati potrebbe essere messa a rischio (*data corruption*)
  - quando il *signal handler* ritorna, la normale sequenza di istruzioni viene ripresa e, nel caso le strutture dati siano state lasciate in uno stato incoerente, si potrebbe incorrere in un *software bug*
- Esempio di potenziale problema:
  - Cosa succede se un segnale viene catturato durante l'esecuzione di una **`malloc`** (che gestisce lo heap), e il signal handler invoca una chiamata a **`malloc`**?
  - In generale, può succedere di tutto, perche' **`malloc`** utilizza una lista linkata di tutte le sue aree allocate e l'interruzione potrebbe avvenire nel mezzo dell'aggiornamento della lista …
- Normalmente, ciò che puo' accadere è un **segmentation fault.**

Funzioni *rientranti* - I

- **Cos'e' una funzione rientrante?**
  - Una funzione rientrante e' una funzione che puo' essere chiamata da piu' di un task concorrentemente senza il rischio di causare un *data corruption*.
  - Viceversa, una funzione non-rientrante e' una funzione che non puo' essere condivisa da piu' di un task, a meno che non si assicurato una mutua esclusione mediante semafori o disabilitazione degli interrupt durante nelle sezioni critiche del codice.
- **Un funzione rientrante:**
  - Non mantiene dati statici per chiamate successive
  - Non ritorna un puntatore a dati statici
  - Utilizza dati locali o garantisce la protezione dei dati globali mediante copie locali degli stessi
  - Non effettua chiamate a qualsiasi altra funzione non-rientrante

Funzioni *rientranti* - II

- **POSIX.1 garantisce che un certo numero di funzioni di libreria siano rientranti:**
  - **`_exit`, `access`, `alarm`, `chdir`,`chmod`, `chown`, `close`, `creat`, `dup`, `dup2`, `execle`, `execve`, `exit`, `fcntl`, `fork`, `fstat`, `get*id`, `kill`, `link`, `lseek`, `mkdir`, `mkfifo`, `open`, `pathconf`, `pause`, `pipe`, `read`, `rename`, `rmdir`, `set*id`, `sig`*, `sleep`, `stat`, `sysconf`, `time`, `times`, `umask`, `uname`, `unlink`, `utime`, `wait`, `waitpid`, `write`**
- **Se una funzione manca nella lista, e' dovuto a:**
  - utilizzo di strutture dati statiche
  - chiamate a **`malloc`** e **`free`**
  - appartenenza alla libreria standard di I/O

Funzioni *rientranti* - III

- **In ogni caso:**
  - Le funzioni rientranti elencate in precedenza possono modificare la variabile **`errno`**
  - Un signal handler che chiama una di quelle funzioni dovrebbe salvare il valore di **`errno`** prima della funzione e ripristinarlo dopo
  - Evitare l'utilizzo di **`printf`** (o altre funzioni non rientranti) nel signal handler

Standard POSIX - I

- **Nelle prime versioni di UNIX**
  - I segnali **non** erano affidabili
    - Potevano andare persi (un segnale viene lanciato senza che un processo ne sia al corrente)
    - Problema derivante in parte dal fatto che, una volta catturato, il signal catcher doveva essere ristabilito

```
signal(SIGINT, sig_int);
void sig_int() {
  signal(SIGINT, sig_int);
  /* process the signal */ }
```

  - Race condition tra l'arrivo del segnale e l'invocazione della `signal()` in `sig_int()`

Standard POSIX - II

- **Cosa succede se un processo riceve un segnale durante una system call?**
  - normalmente, l'eventuale azione associata viene eseguita solo dopo la terminazione della system call
  - in alcune system call "lente", le prime versioni di Unix potevano interrompere la system call, la quale ritornava –1 come errore e la variabile **`errno`** veniva settata a EINTR

Standard POSIX - III

- **Motivazioni:**
  - in assenza di interruzioni da segnali:
    - una lettura da terminale resta bloccata per lunghi periodi di tempo
    - un segnale di interruzione non verrebbe mai consegnato
  - poiché il processo ha catturato un segnale, c'è una buona probabilità che sia successo qualcosa di significativo

Standard POSIX - IV

- **System call "lente":**
  - operazioni **`read`** su file che possono bloccare il chiamante per un tempo indeterminato (terminali, pipe, connessioni di rete)
  - operazione **`write`** su file che possono bloccare il chiamante per un tempo indeterminato prima di accettare dati
  - **`pause`**, **`wait`**, **`waitpid`**
  - certe operazioni **`ioctl`**
  - alcune system call per la comunicazione tra processi
- **Problemi:**
  - bisogna gestire esplicitamente l'errore dato dalle interruzioni

Standard POSIX - V

- **Esempio di gestione:**

```
while ( (n= read(fd, buff, BUFFSIZE)) < 0 )
  { if (errno != EINTR) break; }
```

- **Restart automatico di alcune system call:**
  - Alcune system call possono ripartire in modo automatico:
    - per evitare la necessita' di gestire l'errore dovuto ad una interruzione (come nell'esempio sopra)
    - perché in alcuni casi non è dato sapere se il file su cui si opera può bloccarsi indefinitamente
  - System call con restart (**`ioctl`**, **`read`**, **`write`**) - solo quando operano su fd che possono bloccarsi indefinitamente
  - System call senza restart (**`wait`**, **`waitpid`**) - sempre

Standard POSIX - VI

- **POSIX e S.O. moderni:**
  - Capacità di interrompere 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 affidabili - I

- **Alcune definizioni:**
  - Diciamo che un segnale è *generato* per un processo quando accade l'evento associato al segnale
    - Esempio: riferimento memoria non valido $\Rightarrow$ **`SIGSEGV`**
    - Quando il segnale viene generato, viene settato un flag nel process control block del processo
  - Diciamo che un segnale è *consegnato* ad un processo quando l'azione associata al segnale viene intrapresa
  - Diciamo che un segnale è *pendente* nell'intervallo di tempo che intercorre tra la generazione del segnale e la consegna

Segnali affidabili - II

- **Bloccare i segnali**
  - Un processo ha l'opzione di bloccare la consegna di un segnale per cui l'azione di default non è configurata a *ignore*
  - Il segnale rimane pending fino a quando:
    - il processo sblocca il segnale
    - il processo cambia l'azione associata al segnale al valore *ignore*
  - La lista dei segnali pending e' restituita da **`sigpending()`**
- **Cosa succede se un segnale bloccato viene generato più volte prima che il processo sblocchi il segnale?**
  - POSIX non specifica se i segnali debbano essere accodati oppure se vengano consegnati una volta sola

Segnali affidabili - III

- **Cosa succede se segnali diversi sono pronti per essere consegnati ad un processo?**
  - POSIX non specifica l'ordine in cui devono essere consegnati
  - POSIX suggerisce che segnali importanti (come **`SIGSEGV`**) siano consegnati prima di altri
- **Maschera dei segnali:**
  - Ogni processo ha una maschera di segnali che specifica quali segnali sono attualmente bloccati
  - E' possibile pensare a questa maschera come ad un valore numerico con un bit per ognuno dei possibili segnali
  - E' possibile esaminare la propria maschera utilizzando la system call **`sigprocmask`**

Gestione segnali - I

- **System call:** `int sigpending(sigset_t *set);`
- **Descrizione:**
  - Ritorna l'insieme di segnali che sono attualmente pending per il processo corrente
- **Esempio:**

```
void pr_mask() {
   sigset_t  sigset;
   int errno_save = errno;
   if (sigpending(&sigset) < 0)
       perror("sigpending error");
   if (sigismember(&sigset, SIGINT))  printf("SIGINT ");
   if (sigismember(&sigset, SIGQUIT)) printf("SIGQUIT ");
       /* remaining signals can go here */
   printf("\n");
   errno = errno_save;
}
```

Gestione segnali - II

```
int sigaction(int signo, struct sigaction
        *newact, struct sigaction *oldact);
```

- Permette di esaminare e/o modificare l'azione associata ad un segnale
- Nei sistemi moderni, **`signal`** è implementata utilizzando **`sigaction`**
- Argomenti:
  - **`signo`** segnale considerato
  - **`newact`** se diverso da **`NULL`**, struttura dati contenente informazioni sulla nuova azione
  - **`oldact`** se diverso da **`NULL`**, struttura dati contenente informazioni sulla vecchia azione

Gestione segnali - III

- **Struttura `sigaction`:**

```
struct sigaction {
  void (*sa_handler)();  /* signal handler   */
  sigset_t sa_mask;     /* addit.block mask */
  int sa_flags;              /* options          */
}
```

- **Descrizione:**
  - **`sa_handler`** è il puntatore all'azione per il segnale (un signal handler, **`SIG_IGN`** o **`SIG_DFL`**)
  - **`sa_mask`** è un insieme addizionale di segnali da bloccare quando un segnale viene catturato da un signal handler
  - **`sa_flags`** descrive flag addizionali per vincolare il comportamento del sistema

Gestione segnali - IV

- **Utilizzo di `sa_mask`**
  - All'inizio dell'esecuzione di un signal handler:
    - il valore corrente della **`procmask`** viene salvato
    - alla **`procmask`** vengono aggiunti
      - i segnali specificati in **`sa_mask`**
      - il segnale specificato da **`signo`**
  - Al termine dell'esecuzione di un signal handler:
    - la **`procmask`** viene ripristinata al valore salvato
- **Alcuni valori per `sa_flag` (non standard POSIX):**
  - **`SA_RESTART`** forza automatic restart
  - **`SA_INTERRUPT`** elimina automatic restart

Gestione segnali - V

- **System call:**

```
int sigsuspend(sigset_t *sigmask);
```

  - La procmask viene posta uguale al valore puntato da **`sigmask`**
  - Il processo è sospeso fino a quando:
    - un segnale viene catturato
    - un segnale causa la terminazione di un processo
  - Ritorna sempre –1 con **`errno`** uguale a **`EINTR`**

Esercizio 3

- Re-implementare l'esercizio 2 utilizzando **`sigaction()`** invece di **`signal()`**
- Suggerimenti:
  - Inizializzare la struttura con 0
  - Assegnare solo i valori indispensabili (e.g., l' interrupt handler)
  - Verificare il codice di ritorno di **`sigaction()`**

Esercizio 4 (per casa)

- Alla fine del modulo 06, l'esercitazione 3.B, proponeva il seguente esercizio:

  Sviluppare un propria implementazione della funzione **`system()`** *(si consulti* **`man 3 system`***), rispettandone il prototipo e la funzionalita'; la gestione dei segnali definita per la* **`system()`** *puo' essere ignorata.*

  Aggiungere all'implementazione della **`system()`** la corretta gestione dei segnali, considerando che nelle specifiche della **`system()`** si richiede che **`SIGINT`** e **`SIGQUIT`** vengano ignorati e che **`SIGCHLD`** venga bloccato.