Segnali

- I segnali in Unix sono un meccanismo semplice per inviare degli interrupt software ai processi.
- Solitamente sono utilizzati per gestire errori e condizioni anomale, piuttosto che per trasmettere dati.
- Relativamente ai segnali, un processo può compiere tre tipi di azioni:
  1. eseguire una opportuna funzione per trattare l'errore (signal handling);
  2. bloccare il segnale;
  3. inviare il segnale ad un altro processo.
- I segnali sono definiti da delle costanti simboliche dichiarate nel file header `<signal.h>`.
- Per la maggior parte dei segnali, i processi eseguono una terminazione normale non appena li ricevono.
- Alcuni segnali (`SIGABRT`, `SIGBUS`, `SIGSEGV`, `SIGQUIT`, `SIGILL`, `SIGTRAP`, `SIGSYS`, `SIGXCPU`, `SIGXFSZ`, `SIGFPE`) invece provocano la terminazione anormale del processo a cui sono indirizzati.

Terminazione normale o anormale

In <sys/wait.h> sono definite delle macro per determinare la causa della terminazione di un processo figlio:

```
#include <sys/wait.h>
.
.
if((pid = wait(&status)) == -1) {
  perror("wait fallita");
  exit(1);
}

if(WIFEXITED(status)) {     /* test di uscita normale */
  exit_status = WEXITSTATUS(status);
  printf("L'exit status del processo %d e' %d\n", pid, exit_status);
}

if(WIFSIGNALED(status)) {   /* test di terminazione su ricezione di un segnale */
  sig_no = WTERMSIG(status);
  printf("Il segnale numero %d ha terminato il processo %d\n", sig_no, pid);
}

if(WIFSTOPPED(status)) {   /* test per vedere se l'esecuzione e' stata interrotta */
  sig_no = WSTOPSIG(status);
  printf("Il segnale numero %d ha interrotto l'esecuzione del processo %d\n", sig_no, pid);
}
```

Signal handling

Per definire come gestire un particolare segnale, si utilizza la system call `sigaction`:

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
```

dove:

- `signo` è il segnale che si vuole gestire;
- `act` specifica come deve essere trattato il segnale; la struttura a cui punta questo parametro è definita come segue:

```
struct sigaction {
  void (*sa_handler)(int); /* azione da compiere */
  sigset_t sa_mask;        /* segnali ulteriori da bloccare */
  int sa_flags;            /* flag che influiscono sull'effetto del segnale */
  void (*sa_sigaction)(int, siginfo_t *, void *); /* handler usato in
                                                   * alternativa
                                                   * a sa_handler se
                                                   * flags vale SA_SIGINFO
                                                   */
}
```

- in `oact` vengono salvati i valori attuali (in modo da poterli eventualmente ripristinare in seguito).

Esempio di signal handling

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

main() {
  static struct sigaction act;
  void catchint(int); /* prototipo della funzione per la gestione del segnale SIGINT */
  act.sa_handler = catchint;  /* registrazione dell'handler */
  sigfillset(&(act.sa_mask)); /* eventuali altri segnali saranno ignorati
                                 durante l'esecuzione dell'handler */
  sigaction(SIGINT, &act, NULL);
  printf("sleep call #1\n");
  sleep(1);
  printf("sleep call #2\n");
  sleep(1);
  printf("sleep call #3\n");
  sleep(1);
  printf("sleep call #4\n");
  sleep(1);
  printf("Terminazione\n");
  exit(0);
}

void catchint(int signo) {
  printf("CATCHINT: signo=%d\n", signo);
  printf("CATCHINT: ritorno al main\n\n");
}
```

Inviare segnali

Per inviare il segnale codificato da `sig` al processo con pid `pid`, si usa la system call `kill`:

```
#include <sys/types.h>
#include <signal.h>

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

La system call `raise` è utile quando si vuole inviare un segnale al processo correntemente in esecuzione:

```
#include <signal.h>

int raise(int sig);
```

Per impostare un timer (ad esempio per l'esecuzione di operazioni che, in certe condizioni, possono bloccare un processo) e generare un segnale `SIGALRM` allo scadere del timer stesso, si può usare la system call `alarm`:

```
#include <unistd.h>

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

Semafori

Il concetto di semaforo è stato introdotto da E.W. Dijkstra per risolvere il problema della sincronizzazione fra processi. Un semaforo è una variabile intera sem su cui sono ammesse le seguenti operazioni:

1. p(sem) (o wait(sem)):

```
if(sem!=0)
  sem=sem-1;
else
  /* attendi finche' sem!=0 e decrementa sem di un'unita' */
```

2. v(sem) (o signal(sem)):

```
sem=sem+1;
/* fai ripartire il primo fra i processi in coda d'attesa */
```

Le due operazioni precedenti sono **atomiche**, i.e., soltanto un processo può alterare il valore di un semaforo ad un dato istante.
L'invariante soddisfatta dai semafori è la seguente:
valore iniziale del semaforo + num. di operazioni v - num. di operazioni complete p ≥ 0

Creazione di semafori

In Unix le operazioni sui semafori sono implementate in modo da lavorare su **insiemi** di semafori. `semget` viene usata per creare un nuovo insieme di semafori:

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int permflags);
```

dove:

- `key` è un identificatore (numero) che deve essere unico in modo da permettere l'utilizzo della risorsa corrispondente (l'insieme di semafori) in processi distinti;
- `nsems` è il numero dei semafori che verranno creati;
- `permflags` determina l'azione compiuta da `semget`; due sono le costanti tipicamente utilizzate:
  - `IPC_CREAT`: crea un insieme di semafori corrispondente al valore `key` (se non esiste già);
  - `IPC_EXCL`: se usato congiuntamente con `IPC_CREAT` provoca il fallimento di `semget` se un insieme di semafori è già associato al valore `key`.

  I 9 bit di `permflags` meno significativi possono essere utilizzati per stabilire i permessi di accesso all'insieme di semafori.

Semafori

Ogni semaforo dell'insieme ottenuto da `semget` si ritrova associate le seguenti informazioni:

- `semval`: il valore del semaforo (intero positivo): non può essere manipolato direttamente, ma soltanto attraverso le system call apposite;
- `sempid`: il PID del processo che ha operato per ultimo sul semaforo;
- `semcnt`: numero di processi in attesa che il semaforo assuma un valore maggiore di quello attuale;
- `semzcnt`: numero di processi in attesa che il semaforo assuma il valore `0`.

Inizializzazione di semafori

L'inizializzazione di un semaforo si effettua per mezzo della seguente chiamata di sistema:

```
#include <sys/sem.h>

int semctl(int semid, int sem_num, int command, union semun ctl_arg);
```

dove:

- `semid` è il valore restituito dalla chiamata a `semget`;
- `sem_num` indica il semaforo su cui compiere l'operazione specificata da `command` nel caso che quest'ultimo indichi una funzione applicabile ad un singolo semaforo piuttosto che a tutto l'insieme;
- `command` specifica l'operazione da portare a termine;
- `ctl_arg` è una `union` definita come segue:

```
union semun {
  int val;
  struct semid_ds *buf;
  unsigned short *array;
};
```

Codici di funzioni per `semctl`

Funzioni IPC standard :

- `IPC_STAT`: memorizza l'informazione dello stato in `ctl_arg.buf`;
- `IPC_SET`: imposta l'ownership ed i permessi secondo i valori in `ctl_arg.buf`;
- `IPC_RMID`: rimuove l'insieme di semafori dal sistema.

Operazioni su singoli semafori :

- `GETVAL`: restituisce `semval`;
- `SETVAL`: imposta `semval` al valore contenuto in `ctl_arg.val`;
- `GETPID`: restituisce `sempid`;
- `GETNCNT`: restituisce `semncnt`;
- `GETZCNT`: restituisce `semzcnt`.

Operazioni su tutti i semafori dell'insieme :

- `GETALL`: memorizza tutti i `semval` in `ctl_arg.array`;
- `SETALL`: imposta tutti i `semval` ai valori contenuti in `ctl_arg.array`.

Operazioni sui semafori (I)

La seguente system call permette che un insieme di operazioni venga eseguito in modo **atomico** su un insieme di semafori:

```
#include <sys/sem.h>

int semop(int semid, struct sembuf *op_array, size_t num_ops);
```

dove:

- `semid` è il valore restituito dalla chiamata a `semget`;
- `op_array` è un array di strutture `sembuf` i cui membri sono:
  - `unsigned short sem_num`: indice di un semaforo dell'insieme;
  - `short sem_op`: codice dell'operazione da compiere;
  - `short sem_flg`: flag che viene solitamente impostato a `SEM_UNDO` per ripristinare lo stato iniziale del semaforo al termine del processo corrente;
- `num_ops` è il numero di strutture `sembuf` presenti nell'array `op_array`.

Operazioni sui semafori (II)

A seconda del valore di `sem_op` si hanno i seguenti casi:

- valore negativo: forma generalizzata della funzione p():

```
if(semval>= ABS(sem_op)) {
  semval=semval-ABS(sem_op);
}
else {
  if(sem_flg & IPC_NOWAIT)
    return -1;
  else {
    /* attendi finche' semval raggiunge o supera ABS(sem_op),
       poi sottrai ABS(sem_op) da semval */
  }
}
```

- valore positivo: forma generalizzata di v(): il valore di `sem_op` viene aggiunto a `semval` e viene mandato in esecuzione uno dei processi in attesa;

- valore `0`: si attende finché `semval` diventa nullo; se `sem_flg` è impostato a `IPC_NOWAIT` e `semval` non vale `0` al momento della chiamata, viene restituito `-1`.

Implementazione di p() e v()

```
int p(int semid) {
  struct sembuf p_buf;
  p_buf.sem_num = 0;
  p_buf.sem_op = -1;
  p_buf.sem_flg = SEM_UNDO;

  if(semop(semid, &p_buf, 1) == -1) {
    perror("p(semid) fallita");
    exit(1);
  }
  return(0);
}

int v(int semid) {
  struct sembuf v_buf;
  v_buf.sem_num = 0;
  v_buf.sem_op = 1;
  v_buf.sem_flg = SEM_UNDO;

  if(semop(semid, &v_buf, 1) == -1) {
    perror("v(semid) fallita");
    exit(1);
  }
  return(0);
}
```

Esempio (I)

Il programma seguente, utilizzando l'implementazione precedente delle primitive p() e v(), genera tre processi figli che utilizzano p() e v() per eseguire delle regioni critiche:

```
#include "pv.h"
int p(int semid);
int v(int semid);
int initsem(key_t semkey);
void handlesem(key_t skey);


main() {
  key_t semkey = 0x200;
  int i;

  for(i=0; i<3; i++) {
    if(fork() == 0)
      handlesem(semkey);
  }
}
```

dove `pv.h` è il seguente file:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>
#define SEMPERM 0600
#define TRUE    1
#define FALSE 0

typedef union _semun {
  int val;
  struct semid_ds *buf;
  ushort *array;
} semun;
```

Esempio (II)

```
int initsem(key_t semkey) {
  int status = 0, semid;

  if((semid = semget(semkey, 1, SEMPERM | IPC_CREAT | IPC_EXCL)) == -1) {
    if(errno == EEXIST)
      semid = semget(semkey, 1, 0);
  }
  else {
    semun arg;
    arg.val = 1;
    status = semctl(semid, 0, SETVAL, arg);
  }

  if(semid == -1 || status == -1) {
    perror("initsem fallita");
    return (-1);
  }

  return semid;
}
```

Esempio (III)

```
void handlesem(key_t skey) {
  int semid;
  pid_t pid = getpid();

  if((semid = initsem(skey)) < 0)
    exit(1);

  printf("\nprocesso %d prima della sezione critica\n", pid);
  p(semid);
  printf("\nprocesso %d all'interno della sezione critica\n", pid);
  sleep(10);
  printf("\nprocesso %d in procinto di abbandonare la sezione critica\n",pid);
  v(semid);
  printf("\nprocesso %d in procinto di terminare\n", pid);
  exit(0);
}
```

Esempio (IV)

Un esempio di esecuzione del programma è il seguente:

```
processo 3361 prima della sezione critica

processo 3361 all'interno della sezione critica

processo 3362 prima della sezione critica

processo 3363 prima della sezione critica

processo 3361 in procinto di abbandonare la sezione critica

processo 3361 in procinto di terminare

processo 3362 all'interno della sezione critica

processo 3362 in procinto di abbandonare la sezione critica

processo 3362 in procinto di terminare

processo 3363 all'interno della sezione critica

processo 3363 in procinto di abbandonare la sezione critica

processo 3363 in procinto di terminare
```

Esercizio

Modificare il programma del primo esercizio della Lezione 15 in modo da implementare l'accesso esclusivo al file delle prenotazioni tramite regioni critiche gestite da semafori.