Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2017-18

Pietro Frasca

Lezione 12

Martedì 14-11-2017

System call per l'uso dei segnali

- Un processo che riceve un segnale può gestire l'azione di risposta alla ricezione di tale evento utilizzando le system call sigaction() o *signal()*.
- La chiamata ***sigaction()*** appartenente allo standard POSIX è più complessa ed è da preferirsi alla *signal(),* definita nello *standard C*.
- Tuttavia, per semplicità, analizzeremo la *signal().*

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

  - ***sig*** è un intero (o la costante simbolica) che specifica il segnale da gestire;
  - ***handler*** è il puntatore alla funzione che implementa il codice da eseguire quando il processo riceve il segnale. Il parametro handler può specificare la funzione di gestione dell'interruzione *(handler),* oppure assumere il valore:

    - ***SIG_IGN*** nel caso in cui il segnale debba essere ignorato;
    - ***SIG_DFL*** nel caso in cui debba essere eseguita l'azione di default.
- la funzione ***handler()*** ha un parametro di tipo intero che, al momento della sua attivazione, assumerà il valore dell'identificativo del segnale che ha ricevuto.
- Le associazioni tra segnali e azioni sono registrate nel PCB del processo.
- Poiché la ***fork()*** copia parte delle informazioni del PCB del padre, comprese quelle riguardanti i segnali e rispettivi handler, nel PCB del figlio, e che padre e figlio condividono lo stesso codice, il figlio eredita dal padre le informazioni relative alla gestione dei segnali e quindi:
  - **Le azioni di default dei segnali del figlio sono le stesse del padre;**
  - **ogni processo figlio ignora i segnali ignorati dal padre;**
  - **ogni processo figlio gestisce i segnali con le stesse funzioni usate dal padre;**

- Dato che padre e figlio hanno *PCB* distinti, eventuali chiamate *sigaction()* o *signal()* eseguite dal figlio sono indipendenti dalla gestione dei segnali del padre.
- Inoltre, un processo quando chiama una funzione della famiglia *exec()* non mantiene l'associazione segnale/handler dato che una *exec()* mantiene parte delle informazioni del PCB del processo che la chiama, ma **non dati e codice** e quindi neanche le funzioni di gestione dei segnali.
- L’esempio seguente mostra l’uso della system call *signal()*.

```
#inc1ude <signal.h>
void gestore (int signum){
  printf("Ricevuto il segnale %d \n", signum);
  /* In alcune versioni di unix l'associazione
   segnale/gestore non è persistente. In questo caso è
   necessario rieseguire la signal.
  */
  // signal(SIGUSR1, gestore);
}
main () (
  signal (SIGUSRl, gestore) ;
  /* da qui in poi il processo eseguirà la funzione gestore
   quando riceverà il segnale SIGUSRl */
  ..............
  signal (SIGUSRl, SIG_IGN) ;
  / * SIGUSRl è da qui ignorato: il processo
      non eseguirà più la funzione gestore in risposta a
       SIGUSRl */
}
```

Invio di segnali tra processi

- I processi possono inviare segnali ad altri processi con la system call ***kill()***:

```
#include <signal.h>
int kill (int pid, int sig);
```

  - ***pid*** è il pid del processo destinatario del segnale ***sig***. Se ***pid*** vale **zero**, il segnale ***sig*** viene inviato a tutti i processi della gerarchia del processo mittente.
  - ***sig*** è il segnale da inviare, espresso come numero o come costante simbolica.

- L'esempio seguente mostra l'uso di *signal()* e *kill()*. Il programma, genera due processi (padre e figlio). Entrambi i processi gestiscono il segnale SIGUSR1 mediante la funzione gestore: il figlio, infatti, eredita l'impostazione della signal del padre chiamata prima della *fork()*. Una volta attivi entrambi i processi, il padre invia continuamente il segnale SIGUSR1 al figlio.

```
#include <stdio.h>
#include <signal.h>
void gestore (int signum) {
   static cont=0;
   printf ("Processo con pid %d; ricevuti n. %d segnali %d
            \n",  getpid(), cont++, signum);
}
int  main () {
  pid_t pid;
  signal(SIGUSR1, gestore);
  pid = fork ();
  if (pid==0) /* figlio */
    for (; ;) pause();
  else /* padre */
    for ( ; ;) {
      kill (pid, SIGUSR1);
      sleep(1);
     }
}
```

- Oltre alla system call *kill()*, esistono altre chiamate di sistema che automaticamente inviano segnali. Ad esempio la funzione ***alarm()*** causa l'invio del segnale ***SIGALRM*** al processo che la chiama, dopo un intervallo di tempo specificato nell'argomento della funzione.

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds)
```

- L'esempio seguente mostra l'uso di ***alarm()*** e ***pause()***. Dopo **ns** secondi viene inviato un segnale di allarme (SIGALRM) e viene eseguita la funzione ***azione()*** specificata in ***signal()***. Il tempo di allarme ns viene incrementato dopo ogni chiamata di *alarm()*.
- La function ***system()*** manda in esecuzione il programma specificato nell'argomento. Nell'esempio viene eseguita la funzione ***system()*** che consente di mandare in esecuzione un programma specificato nell'argomento, in questo caso viene eseguito l'utility **date** che visualizza data e ora correnti.

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int ns=1; // periodo iniziale di allarme (1 secondo)
int nmax=10; // valore massimo dell'intervallo di allarme
void azione(){
  /* questa funzione viene eseguita ogni volta
     che il processo riceve il segnale SIGALRM,
  */
  printf("Segnale di allarme ricevuto...eseguo date \n");
  system("date"); // esegue il comando date

  /*
    riassegnamento del periodo di allarme
    che cancella il precedente periodo assegnato.
  */
  alarm(ns); // ns viene incrementato
}
```

```
int main() {
  int i;
  signal(SIGALRM,azione);
  alarm(ns);
  while(ns <= nmax) {
    printf("processo in pausa\n");
    pause();
    printf("fine pausa\n");
    ns++;// incremento del periodo di allarme
   }
   exit(0);
}
```

Sincronizzazione tra thread in POSIX

- Per risolvere problemi di mutua esclusione la libreria pthread definisce il **mutex, i semafori** e le **variabili condition**.
- Il mutex è un particolare semaforo il cui valore intero (lo stato del mutex) può essere:

  0 (**occupato**) oppure

  1 (libero).

- Generalmente un mutex è usato per garantire che gli accessi a una risorsa condivisa avvengano in mutua esclusione.
- Nella libreria pthreads il mutex è definito dal tipo **pthread_mutex_t** che rappresenta:
  - **lo stato del mutex**;
  - **la coda di thread** nella quale saranno sospesi i thread in attesa che il mutex sia libero.

- Per definire un mutex *M*:

```
pthread_mutex_t M;
```

- Una volta definito, un mutex, si inizializza con delle proprietà.
- La funzione per l'inizializzazione è **pthread_mutex_init**, la cui sintassi è:

```
int pthread_mutex_init(
    pthread_mutex_t *M,
    const pthread_mutexattr_t *attr)
```

dove

  - ***M*** è il mutex da inizializzare e
  - **attr** punta a una struttura che contiene gli attributi del mutex; se il valore di *attr* è **NULL**, il mutex viene inizializzato con i valori di default (con stato posto a ***libero***).

- Sul mutex sono possibili le operazioni: ***lock()*** e ***unlock()***, che sono concettualmente equivalenti rispettivamente alle funzioni ***wait()*** e ***signal()*** dei semafori.
- In particolare, la ***pthread_mutex_lock()*** è la realizzazione della ***wait()*** per il mutex:

```
int pthread_mutex_lock (pthread_mutex_t *M);
```

dove ***M*** rappresenta il mutex.

Se *M* è *occupato,* il thread chiamante si sospende nella coda associata al mutex; altrimenti *acquisisce M* e continua l'esecuzione.

- La ***pthread_mutex_trylock()*** è la versione non bloccante della *lock()*:

```
int pthread_mutex_trylock (pthread_mutex_t *M);
```

Si comporta come la *pthread_mutex_lock()*, tranne che nel caso di mutex già occupato non blocca il thread chiamante e ritorna immediatamente con l'errore **EBUSY**

- La system call:

  ```
  int pthread_mutex_unlock(pthread_mutex_t *M);
  ```

  è la realizzazione della ***signal()***.
- Infatti, L'effetto della *_unlock()*, dipende dallo stato della coda di thread associata al mutex: se ci sono thread in attesa del mutex, ne risveglia uno, altrimenti libera il mutex.

- La system call:

  ```
  int pthread_mutex_destroy (pthread_mutex_t *M);
  ```

  dealloca tutte le risorse assegnate al mutex *M*.

Esempio 1 - mutex

```
/* Il tread main iniziale crea due thread di nome Th1 e
   Th2. I due thread condividono la variabile intera A alla
   quale inizialmente è assegnato il valore 10. Il thread
   Th1 incrementa di 1 il valore di A e attende 1 secondo,
   mentre Th2 decrementa di 1 il valore di A e attende 1
   secondo. Entrambi i thread eseguono le operazioni
   suddette per 10 volte, quindi terminano.
*/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int A=10; // variabile condivisa
pthread_mutex_t M; /* mutex per mutua esclusione */
```

```
void *codice_Th1 (void *arg){
  int i;
  for (i=0; i<10;i++){
    printf("Thread %s: ", (char *)arg);
    pthread_mutex_lock(&M); /* prologo sez. critica */
    A++;
    printf (" A = %d \n",A);
    pthread_mutex_unlock(&M); /* epilogo sez. critica */
    sleep(1);
  }
  pthread_exit(0);
}
```

```
void *codice_Th2 (void *arg){
  int i;
  for (i=0; i<10;i++){
    printf("Thread %s: ", (char *)arg);
    pthread_mutex_lock(&M); /* prologo sez. critica */
    A--;
    printf (" A = %d \n",A);
    pthread_mutex_unlock(&M); /* epilogo sez. critica */
    sleep(1);
  }
  pthread_exit(0);
}
```

```
int main(){
  pthread_t th1, th2;

   // creazione e attivazione del primo thread
  if (pthread_create(&th1,NULL,codice_Th1, "th1")!=0){
    fprintf(stderr,"Errore di creazione thread 1 \n");
    exit(1);
  }

   // creazione e attivazione del secondo thread
  if (pthread_create(&th2,NULL,codice_Th2, "th2")!=0){
    fprintf(stderr,"Errore di creazione thread 2 \n");
    exit(1);
  }
```

```
  // attesa della terminazione del primo thread
  if (pthread_join(th1,NULL) !=0)
    fprintf(stderr,"join fallito %d \n",ret);
  else
    printf("terminato il thread 1 \n");

  // attesa della terminazione del secondo thread
   if (pthread_join(th2,NULL) !=0)
    fprintf(stderr,"join fallito %d \n",ret);
  else
    printf("terminato il thread 2 \n");
  return 0;
}
```

Esempio 2 - mutex

```
/* Il thread main crea una matrice di numeri interi di
dimensione NxM assegnando a ciascun elemento della matrice
un valore casuale compreso tra 0 e 255.

      Dopo aver creato la matrice, il thread main crea N
thread figli ciascuno dei quali ha il compito di eseguire
la somma di una riga della matrice.

      Ciascun thread aggiunge la somma che ha calcolato
ad una variabile di nome sommaMat che al termine
dell'esecuzione del programma conterrà la somma di tutti
gli elementi della matrice. Il valore della variabile
sommaMat deve essere stampato su video dal thread main. */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define N 20
#define M 1024
```

```
pthread_mutex_t mut; /* mutex condiviso tra threads */
int a[N][M];
int sommaMat=0;

void *sommaRiga_th(void *arg){
   int i=(int)arg;
   int j;
   int sommaRiga=0;
   for(j=0;j<M;j++)
       sommaRiga+=a[i][j];
   pthread_mutex_lock(&mut); /* prologo sez. critica */
   sommaMat += sommaRiga;
   printf("thread %d: sommaRiga=%d
       somma=%d\n",i,sommaRiga,sommaMat);
   //sleep(1);
   pthread_mutex_unlock(&mut); /* epilogo sez. critica */
   pthread_exit(0);
 }
```

```
int main () {
  int i,j;
  pthread_t th[N];
  pthread_mutex_init (&mut,NULL);
  srand(time(NULL)); // seme per random
  for (i=0;i<N;i++)
    for (j=0;j<M;j++)
      a[i][j]=rand()%256;
   for (i=0;i<N;i++)
    if (pthread_create(&th[i],NULL,sommaRiga_th,(int *)i)
       !=0) {
      fprintf (stderr, "errore create thread i \n");
      exit(1);
     }
  for (i=0;i<N;i++)
    pthread_join(th[i],NULL);
  printf("Somma = %d \n",sommaMat);
}
```

Semafori

- POSIX fornisce due tipi di semaforo: con nome o senza nome. Vedremo l'uso di semafori senza nome.

```
#include <semaphore.h>
sem_t sem;
int sem_init(sem_t *sem, int pshared,
             unsigned value);
```

La funzione ***sem_init()***, crea e inizializza il semaforo, ha tre parametri:

- *sem*, un puntatore al semaforo;
- *pshared,* un intero che indica il **livello di condivisione**;
- *value*, un intero che indica il valore iniziale del semaforo,

Se il livello di condivisione (secondo parametro) vale 0, il semaforo è condivisibile solo dai thread che appartengono allo stesso processo che ha creato il semaforo. Un valore diverso da 0, consente l'accesso al semaforo anche da parte di altri processi. Ad esempio,

```
sem_init(&sem, 0, 4);
```

crea un semaforo e lo inizializza al valore 4.

In questa libreria, le primitive *wait()* e *signal()* prendono rispettivamente i nomi di ***sem_wait()*** e ***sem_post()***.

Altre utili funzioni sono: ***sem_trywait()*** che è la versione non bloccante di *sem_wait()*; ***sem_getvalue()*** che restituisce il valore corrente di un semaforo; ***sem_destroy()*** che dealloca le risorse allocate per il semaforo;

Il frammento di codice seguente mostra l'uso del semaforo.

```
#include <semaphore.h>

   sem_t sem;
   …
   /* crea il semaforo */
   sem_init(&sem, 0, 1);
   …

   sem_wait (&sem); /* acquisisce il semaforo */

   /* SEZIONE CRITICA */

   sem_post(&sem); /* rilascia il semaforo */
   …
```