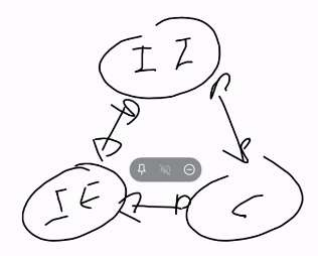
Gestione Eventi 2 UNIX

martedì 25 novembre 2025 17:44

Vediamo ora come avvengono le segnalazioni nei sistemi UNIX : ovvero le relazioni tra un'applicazione attiva ed uno specifico evento che il sw del so è in grado di rilevare. In generale o meglio in qualunque momento si hanno 3 scenari di reazione:

1. Ignorare la segnalazione implicitamente
   a. Fatto fino ad ora
2. Ignorare la segnalazione esplicitamente
3. Catturare la segnalazione



Il cui passaggio da uno stato all'altro è stato fatto da una system call (cambio stato vs la segnalazione). in dettaglio :

SIGHUP (1): Hangup. Il processo riceve questo segnale quando il terminale a cui era associato viene chiuso (ad esempio nel caso di un xterm)

SIGINT (2): Interrupt. Il processo riceve questo segnale quando l'utente preme la combinazione di tasti di interrupt (solitamente Control+C)

SIGQUIT (3): Quit. Simile a SIGINT, ma in più, **in caso di terminazione del processo**, il sistema genera un "core dump" , ovvero un file che contiene lo stato della memoria al momento in cui il segnale SIGQUIT è stato ricevuto. Solitamente SIGQUIT viene generato premendo i tasti Control+\

SIGILL (4): Illegal Instruction. Il processo riceve questo segnale nel caso in cui tenti di eseguire un'istruzione proibita (o inesistente)

SIGKILL (9): Kill. Questo segnale non può essere catturato in nessun modo dal processo ricevente, **che non può fare altro che terminare**. Mandare questo segnale è il modo per ``uccidere'' un processo

SIGSEGV (11): Segmentation violation. Il processo riceve questo segnale quando tenta di eseguire un accesso non supportato all'interno dello spazio di indirizzamento

SIGTERM (15): Termination. Inviato come richiesta non forzata di terminazione.

SIGALRM (14): Alarm. Inviato allo scadere del conteggio dell'orologio di allarme

SIGUSR1, SIGUSR2 (10, 12): User defined. Non hanno un significato preciso, e possono essere utilizzati per implementare un qualsiasi protocollo di comunicazione e/o sincronizzazione

SIGCHLD (17): Child death. Inviato ad un processo quando uno dei suoi figli termina

Vediamo quindi come :

1. **Inoltrare segnalazione verso destinazione**
   a.

int kill(int pid, int segnale)	
Descrizione	richiede l'invio di un segnale
Argomenti	1) pid: identificatore di processo destinatario del segnale (**ovvero il suo main thread**) 2) segnale: specifica del numero del segnale da inviare
Restituzione	-1 in caso di fallimento

Default al lato destinazione (segnali più importanti)
- tutti i segnali tranne SIGKILL e SIGCHLD sono ignorati implicitamente e al loro arrivo il processo destinatario termina
- SIGCHLD è ignorato implicitamente, ma il suo arrivo non provoca la terminazione del processo destinatario
- SIGKILL non è ignorabile, ed il suo arrivo provoca la terminazione del processo destinatario

2. **Richiesta emissione segnalazione per processo corrente**
   a.

int raise(int segnale)	
Descrizione	richiede l'invio di un segnale al **thread** corrente
Argomenti	segnale: specifica del numero del segnale da inviare
Restituzione	0 in caso di successo

   b. Possibile colpire lo stesso thread
3. **Segnali temporizzati**
   a. Segnale sveglia dopo tot tempo

b.

unsigned alarm(unsigned time)	
Descrizione	invoca l'invio del segnale SIGALRM a se stessi
Argomenti	time: tempo allo scadere del quale il segnale SIGALRM deve essere inviato
Restituzione	tempo restante prima che un segnale SIGALRM invocato da una chiamata precedente arrivi

4. **Catturare/ignorare segnali**
   a. Syscall basica
   b.

void (*signal(int sig, void (*ptr)(int)))(int)	
Descrizione	specifica il comportamento di ricezione di un segnale
Argomenti	1) sig: specifica il numero del segnale da trattare 2) ptr: puntatore alla funzione di gestione del segnale o SIG_DFL o SIG_IGN
Restituzione	SIG_ERR in caso di errore altrimenti il valore della precedente funzione di gestione del segnale che viene sovrascritto con ptr

- SIG_DFL setta il comportamento di default
- SIG_IGN ignora il segnale esplicitamente (importante per il segnale SIGCHLD)

5. **Attesa**
   a. Mette in pausa un thread
   b. Syscall bloccante -> il thread viene messo in blocco fino ad invocazione signal

int pause(void)	
Descrizione	blocca il **thread** in attesa di un qualsiasi segnale
Restituzione	sempre −1 (poiche' e' una system call interotta da segnale)

   c.
- `pause()` ritorna sempre al completamento della funzione di handling del segnale
- non è possibile sapere direttamente da `pause()` (ovvero tramite valore di ritorno) quale segnale ha provocato lo sblocco; si può rimediare facendo si che l'handler del segnale modifichi il valore di qualche variabile globale o thread_local

Vediamo ora l'**ereditarietà**: ovvero il comportamento associato alla ricezione viene ereditato dai figli (fork) , mentre nel caso di exec si mantiene solo il comportamento ignore e default , altrimenti se diverso si ha comunque il default. Vediamo ora il **meccanismo di consegna del segnale** : ricordiamo che si ha cambio del flusso di esecuzione (effettuata dal kernel), il quale al prossimo schedule prova a riportare il thread in user mode (da blocked a ready), il che provoca passaggio da ERRNO a EINTR . Ricordiamoci che nello standard POSIX 1 ERRNO non era thread safe in quanto varabile globale; invece dalla variante POISX1.c questa variabile ha ambito thread local (TLS) . Quindi in generale ERRNO è thread-safe anche in applicazioni multi thread. Vediamo ora degli esempi :

1. **Alarm**
   a.
```c
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>

#define timeout 1

void wakeup(int sgnumber){
        printf("I'm awake\n");
        alarm(timeout);
}

int main (int argc, char**argv){

        signal(SIGALRM, wakeup);
        alarm(timeout);
        while (1){
                pause();
        }

}
```
   b. **Signal -> segnale da attivare con funzione da invocare**
   c. **Il sigint viene ignorato implicitamente**
2. **Race**
   a. Sigint = ctrl+c
   b. Non possibile in sistemi time-sharing
   c.
```c
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>

unsigned int a,b,c,d,e,f;

void printdata(int sgnumber){
        printf("values are %d - %d - %d -%d -%d - %d \n",a,b,c,d,e,f);
}

int main (int argc, char**argv){

        unsigned int i = 0;
        signal(SIGINT, printdata);

        while (1){
                a = b = c = d = e = f = (++i)%1024;
        }

}
```
   d. Se invoco il ctrl+c potrebbe capitare che i valori siano diversi !!

3. **Killer**
   a.
```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

int main(){

   int pid;

   scanf("%d",&pid);
   printf("killing %d - res is %d\n",pid,kill(pid,SIGINT));

}
```
   b. Va a colpire una specifica applicazione riferita dal pid
4. **Signal and thread**
   a. Non si sa quale signal venga chiamata prima in quanto processi concorrenti
   b. Il gestore eseguito è l'ultimo invocato
   c.
```c
#include <stdio.h>
#include <pthread.h>
#include <signal.h>
#include <unistd.h>


void printA(){
        printf("A\n");
}

void printB(){
        printf("B\n");
}

void *generic_threadA(){
        signal(SIGINT,printA);
        while(1) {
                pause();
                printf("I'm thread A\n");
        }
}

void *generic_threadB(){
        signal(SIGINT,printB);
        while(1) {
                pause();
                printf("I'm thread B\n");
        }
}


int main(int argc, char **argv){

  pthread_t thread;
  pthread_attr_t attr;

  pthread_create(&thread, NULL, generic_threadA, NULL);
  pthread_create(&thread, NULL, generic_threadB, NULL);


  while(1) {
        pause();
        printf("I'm thread main\n");
  }


}
```
   d. Se colpisco il main stampa B
   e. Se colpisco altro processo sempre ultima signal
5. **Sigchild**
   a. Se compilo con IGNORE ignoro la terminazione dei processi figli (con eventuale pid)-> non mantenuto dal kernel
   b.
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

int main(int a, char **b){

        int i;

#ifdef IGNORE
        signal(SIGCHLD,SIG_IGN);
#endif

        for (i=0; i<10; i++){
                if(fork() == 0) exit(0);

        }

        pause();

}
```
6. **Signal numbers**
   a. **Generico handler per gestione segnali**
   .
```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>


void generic_handler(int signal){

       printf("received signal is %d\n",signal);
       fflush(stdout);
}
```

b.

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


void generic_handler(int signal){

        printf("received signal is %d\n",signal);
        fflush(stdout);
}



int main(int argc, char **argv){

  int  i;

  signal(SIGINT,generic_handler);
  signal(SIGUSR1,generic_handler);
  signal(SIGUSR2,generic_handler);

  while(1) {
        pause();
  }

}
```