Lezione 10: Segnali

10.1 Introduzione

Un segnale è la notifica ad un processo che si è verificato un evento.

I segnai sono simili alle interruzioni hardware, interrompono il normale flusso di esecuzione di un programma, in molti casi non è possibile predire esattamente quando il segnale arriverà.

Un processo può inviare un segnale ad un altro processo. Usati in questo modo i segnali rappresentano una forma primitiva di IPC, un processo può anche inviare un segnale a se stesso.

La sorgente principale di segnali inviati ad un processo è il kernel.

Tipi di eventi per cui il kernel genera un segnale per un processo:

- Eccezioni HW.
- L'utente digita un carattere speciale del terminale.
- Si è verificato un evento SW.

Ogni segnale è definito come un intero unico, partendo sequenzialmente da 1.

sono definiti in <signal.h> con nomi simbolici della forma SIGxxx

Nei programmi devono essere sempre usati i nomi dimbolici, i numeri effettivi usati per ogni segnale possono variare a seconda delle implementazioni.

10.2 Alcune definizioni

La disposizione per un segnale è l'indicazione al kernel su cosa fare in seguito all'occorrenza del segnale, un segnale è stato generato per un processo quando si verifica l'evento che genera il segnale.

Quando è iintrapresa un'azione in corrispondenza di un determinato segnale si dice che il segnale è stato consegnato.

Nel periodo di tempo che intercorre tra la generazione e la consegna si dice che il segnale è pendente.

Come un segnale generato più di una volta venga consegnato ad un processo, dipende dalla particolare implementazione.

10.3 Segnali

Un processo non si può limitare a controllare se un segnale si è verificato, un processo deve anche comunicare al kernel cosa fare in caso di occorrenza di uno specifico segnale.

Tra le cose che un processo può chiedere al kernel di fare:

- Ignora il segnale: quelli che non vengono ignorati sono SIGKILL e SIGSTOP, altri da non ignorare a livello HW sono SIGFPE, SIGILL e SIGSEGV.
- Intercetta il segnale: fornische una funzione da eseguire per un segnale.
- Eseguire le azioni di default: normalmente è la terminazione del processo.

SIGFPE

Generato quando si divide per 0.

Default terminazione e file core

SIGHUP

Generato quando il terminale di controllo viene chiuso.

Default Terminazione

SIGINT

Generato con CONTROL-C

Default termina il processo

SIGKILL

Viene mandato dal proprietario o da root.

Non deve essere ignorato.

SIGSEGV

Generato quando il processo tenta di accedere a memoria al di fuori del suo segmento

Default termina

SIGUSR1, SIGUSR2

Sono definiti dall-utente

Default Termina

10.4 La funzione signal()

La funzione signal() fornisce lo strumento per istruire il kernel ad eseguire una determinata azione quando il processo chiamante ricecve un determinato segnale. Tale azione può essere:

- Ignorare il segnale (SIG_IGN)
- Far eseguire al kernel l'azione di default definita per tale segnale (SIG_DFL)
- Passare al kernel l'indirizzo di una funzione da eseguire quano si presenta tale funzione.

La funzione ritorna le disposizioni precedenti per il segnale.

```
#include <signal.h>
void(*signal(int signo, void (*func)(int)))(int);
```

Ha due argomenti e ritorna un puntatore ad una funzione che non ritorna nulla.

signo è un intero il nome del segnale occorso

func è ul puntatore ad una funzione che prende come argomento un intero e non ritorna nulla.

Il valore di func è

- SIG_IGN
- SIG_DFL
- indirizzo di una funzione da chiamare quando occorre il segnale

La funzione di cui è ritornato l'indirizzo come valore della funzione signal ha un unico argomento intero.

Il prototipo della funzione signal può essere reso più semplice

```
typedef void(*signalhendler_t)(int);
signalhandler_t signal(int,signalhandler_t);
```

10.4.1 Esempio

```
#include <signal.h>
int main(void){
    singal(SIGINIT, SIG_IGN);
    while(1);
}
```

10.4.2 Gestore che cattura vari segnale e ne stampa il tipo

```
# include <signal.h>
# include "apue.h"
static void sig_usr(int); // un solo handler per tutti

int main(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR1");

    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR2");

    if (signal(SIGINT, sig_usr) == SIG_ERR) // <CTRL C>
```

```
        err_sys("can't catch SIGINT");

    if (signal(SIGTSTP, sig_usr) == SIG_ERR) // <CTRL Z>
        err_sys("can't catch SIGTSTP");
    for ( ; ; ) pause();

    static void
    sig_usr(int signo)// signo è il numero del segnale
    {
        if (signo == SIGUSR1)
            printf("received SIGUSR1\n");
        else if (signo == SIGUSR2)
            printf("received SIGUSR2\n");
        else if (signo == SIGINT)
            printf("received SIGINT\n");
        else if (signo == SIGTSTP)
            printf("received SIGTSTP\n");
        else
            err_dump("received signal %d\n", signo);
        return;
}

}
```

10.5 Gestore dei segnali

L'invocazione di un gestore può interrompere il flusso principale di un programma in qualsiasi momento.

Il kernel invoca il gestore per conto del processo, quando il gestore ritorna l'esecuzione del programma riprende nel punto in cui il gestore lo ha interrotto

Sebbene i gestori possono fare tutto, essi dovrebbero essere il più semplice possibile per evitare race condition.

10.6 Segnali ed avvio dei programmi

Quando è eseguito un programma l'azione associata a ciascun segnale è quella di default o ignora.

Tutti i segnali sono impostati all'azione di default a meno che il processo che invoca exec stia ignorando il segnale.

I processi creati con fork ereditano la disposizione dei segnali dei genitori, iniziano con una copia dell'immagine della memoria del genitore.

Un segnale intercettato in processo non può essere interceettato da un eseguibile avviato con exec, perchè per quest'ultimo l'indirizzamento della funzione di gestione del segnale non ha senso.

10.7 SC kill() e raise()

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
```

Kill permette adu un processo di inviare il segnale signo al processo pid, raise() a se stesso

Ritorna 0 se ok - < 0 in caso di errore.

La chiamata a raise(signo) è equivalente alla chiamata kill(getpid(), signo);

pid può assumere i seguent valori:

- pid > 0 il segnale viene inviato al processo con quel pid quel segnale.
- pid == 0 il segnale viene inviato a tutti i processi con lo stesso process group del processo invocante.
- pid < 0 il segnale viene inviato a tutti i processi con process group ID uguale al modulo di pid.

10.8 SC kill()

POSIX.1 definisce il segnale numero 0 come segnale nullo, serve se un dato processo esiste, inviando un segnale nullo.

Un processo può mandare un segnale ad un altro processo se entrambi sono in esecuzione con gli stessi real o effective user ID, sempre se è in esecuzione con i permessi di superutente.

10.9 Richiedere un segnale di allarme: alarm()

Imposta un timer che quando scade viene generato un segnale che viene notificato al processo che ha impostato il timer.

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

10.10 Attendere un segnale: pause()

non riceve argomenti e sospende il processo fino alla chiamata del processo.

```
#include <unistd.h>
int pause(void)
```

10.11 sleep

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

10.12 nanosleep

Pone il processo in stato di sleep per il tempo specificato da req in caso di interruzione restituisce il tempo restante in rem

Ka funzione restituisce 0 se l'attesa viene completata o -1 in caso di errore, nel qual caso errno assumerà uno dei valori

EINVAL si è specificato un numero di secondi negativo o un numero di nanosecondi maggiore di 999.999.999

EINTR la funzione è stata interrotta da un segnale

La struttura timespec è usata per specificare intervalli di tempo con precisione al nanosecondo. è specificata in <time.h>

10.13 abort()

Invochiamo quando vogliamo inviare un segnale ad un processo che termina in situazioni anomale

```
#include <stdlib.h>
void abort(void)
```

10.14 Segnali inaffidabili

Nelle prime versioni di unix i segnali erano inaffidabili, potevano andare persi: l'occorrenza di un segnale non era nota al processo.

Un processo aveva poco controllo su di un segnale.

Un processo poteva ignorare il segnale o catturarlo, talvolta è necessario dire al kernel di bloccare il segnale ma non di ignorarlo in modo da poterne ricordare l'occorrenza.

10.14.1 problemi(1)

La disposizione per il segnale era azzerata a quella di default ogni volta che il segnale occorreva.

10.14.2 problemi(2)

Il processo non ha la possibilità di bloccare il segnale quando si verifica.

Ci sono casi in cui l'occorrenza di un segnale è necessario prevenirla ma ricordarla.

10.15 Rientranza delle funzioni

Una funzione rientrante, anche nota come funzione "thread-safe", è una funzione che può essere eseguita in modo sicuro da più processi o thread contemporaneamente senza causare problemi di concorrenza o inconsistenza dei dati. In altre parole, una funzione rientrante è progettata in modo tale da garantire che ogni sua chiamata abbia un ambiente di esecuzione indipendente e isolato, senza interferenze con altre chiamate alla stessa funzione da parte di altri processi o thread.

Le problematiche principali che possono rendere una funzione non rientrante sono legate alla gestione delle variabili e delle risorse condivise:

1. Utilizzo di variabili globali: Se una funzione fa uso di variabili globali, queste possono essere soggette a modifiche da parte di altri processi o thread mentre la funzione è in esecuzione. Se non viene gestita correttamente questa concorrenza, si possono verificare situazioni in cui il valore delle variabili diventa inconsistente o imprevisto, portando a risultati errati.
2. Utilizzo di variabili statiche: Le variabili statiche mantengono il loro valore anche dopo che la funzione è stata eseguita e ritornata. Se più istanze della funzione sono in esecuzione contemporaneamente, le variabili statiche possono condividere il loro stato tra di loro, portando a comportamenti non prevedibili.

Un esempio concreto è la funzione `malloc()`, utilizzata nei sistemi operativi per l'allocazione dinamica di memoria. Se più thread chiamano contemporaneamente `malloc()`, potrebbero competere per l'accesso alle risorse di memoria, portando a situazioni di race condition o a corruzione della memoria stessa.

Per garantire la rientranza delle funzioni in un sistema operativo, è necessario adottare pratiche di programmazione sicura, come l'uso di variabili locali o la sincronizzazione delle risorse condivise tramite meccanismi come mutex o

semafori. In questo modo, si evitano problemi di concorrenza e si assicura che il sistema operativo sia stabile e affidabile anche in presenza di molteplici processi o thread in esecuzione simultanea.

10.16 Esempi di uso dei segnali

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdlib.h>
```

10.17 Altri esempi

10.17.1 critical.c

Il programma criticl.c suggerisce come si possono proteggere pezzi di codice da interruzioni dovute a control+c.

Procede nel modo seguente:

- Salva il precedente valore dell'handler indicando che il segnale deve essere ignorato.
- Esegue la regione di codice protetta.
- Ripristina il valore originale dell'handler.

```
#include <stdio.h>
#include <signal.h>
int main(void){
    void(*oldhandler)(int);
    printf("Posso essere interrotto\n");
    sleep(3);
```

```
    oldhandler = signal(SIGINT, SIG_IGN);
    printf("Sono protetto da control+c\n");
    sleep(3);
    signal(SIGINT, oldhandler);
    printf("posso essere interrotto\n");
    sleep(3);
    printf("Ciao\n");
}
```

10.18.2 limit.c

```
void childHandler (int sig) { // Eseguito se il figlio termina
    int childPid, childStat; /* prima del genitore */
    childPid = wait (&childStat); /* Accetta il codice di termi
    printf ("Child %d terminated within %d seconds\n", childPid,
    exit (/* USCITA con SUCCESSO */ 0);
}
```

10.18.3 pulse.c

Questo programma crea due figli che erano in un ciclo infinito e mostrnao un messaggio ogni secondo, il padre aspetta due secondi e quindi sospende il primo filgio, mentre il secondo continua l'esecuzione. Dopo altri 2 secondi il padre riattiva il primo figlio aspetta 2 secondi e poi termina

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

int main (void) {
int pid1;
int pid2;
```

```
pid1 = fork ();

if (pid1 == 0) { /* Primo figlio */
    while (1) { /* Ciclo infinito */
        printf ("pid1 is alive\n");
        sleep (1);
    }
}

pid2 = fork ();
if (pid2 == 0) { /* Secondo figlio */
    while (1) { /* Ciclo infinito */
        printf ("pid2 is alive\n");
        sleep (1);
    }
}
```