

# Segnali

Laboratorio Sistemi Operativi

Giuseppe Salvi

Email: [giuseppe.salvi@uniparthenope.it](mailto:giuseppe.salvi@uniparthenope.it)

# Segnali

- Un segnale è la notifica, ad un processo, che si è verificato un evento.
- I segnali 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 (con i dovuti permessi) 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

# Segnali (cont.)

- La sorgente principale di segnali inviati ad un processo è il kernel
- Tra i tipi di eventi che fanno sì che il kernel generi un segnale per un processo:
  - Eccezione hw: l'hw ha individuato una condizione di errore che notifica al kernel, che a sua volta invia il segnale corrispondente al processo coinvolto
    - Esempi: eseguire un'istruzione macchina mal formata, una divisione per 0 o riferire una parte di memoria inaccessibile
  - L'utente digita un carattere speciale del terminale che genera segnali. Tra questi Control-C (SIGINT), Control-Z (SIGSTOP)
  - Si è verificato un evento software
    - l'input diventa disponibile su un descrittore di file, la finestra del terminale è stata ridimensionata, scaduto un timer, un processo figlio è terminato ecc...

# Segnali (cont.)

- Ogni segnale è definito come un intero (piccolo) unico, partendo sequenzialmente da 1
  - definiti in `<signal.h>` con nomi simbolici della forma **SIGxxxx**
- Nei programmi devono essere sempre usati i nomi simbolici
  - i numeri effettivi usati per ogni segnale possono variare a seconda delle implementazioni
- I segnali ricadono in due categorie
  - i segnali tradizionali o standard, usati dal kernel per notificare gli eventi ai processi
  - segnali real-time

# Alcune definizioni

- La **disposizione** (o azione associata) per un segnale è l'indicazione al kernel su cosa fare in seguito all'occorrenza del segnale
- Un segnale è stato generato (**generated**) per un processo quando si verifica l'evento che genera il segnale
- Quando è intrapresa un'azione in corrispondenza di un determinato segnale si dice che il segnale è stato consegnato (**delivered**)
- Nel periodo di tempo che intercorre tra la generazione del segnale e la sua consegna si dice che il segnale è pendente (**pending**)
- Come un segnale generato più di una volta venga consegnato ad un processo, dipende dalla particolare implementazione

# Elencare i segnali disponibili

```
$ kill -l
```

1) SIGHUP

2) SIGINT

3) SIGQUIT

4) SIGILL

5) SIGTRAP

6) SIGIOT

7) SIGBUS

8) SIGFPE

9) SIGKILL

10) SIGUSR1

11) SIGSEGV

12) SIGUSR2

13) SIGPIPE

14) SIGALRM

15) SIGTERM

17) SIGCHLD

18) SIGCONT

19) SIGSTOP

20) SIGTSTP

21) SIGTTIN

22) SIGTTOU

23) SIGURG

24) SIGXCPU

25) SIGXFSZ

26) SIGVTALRM

27) SIGPROF

28) SIGWINCH

29) SIGIO

30) SIGPWR

```
$
```

# 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
  - Tre sono le cose che un processo può chiedere al kernel di fare (azione associata ad un segnale):
    - **Ignorare il segnale:** (tutti tranne **SIGKILL** e **SIGSTOP**); nel caso si decida di ignorare i segnali generati da eccezioni hardware (**SIGFPE**, **SIGILL**, e **SIGSEGV**), il comportamento del processo è indefinito
    - **Intercettare il segnale:** fornire una funzione da eseguire per un determinato segnale
    - **Eseguire le azioni di default**

# Azione di default di un segnale

- Dopo la consegna di un segnale, un processo esegue un'azione di default associata al segnale
- La tipologia di azione dipende dal segnale
- Nella maggior parte dei casi
  - Il processo è terminato
  - Viene generato un file che contiene un'immagine della memoria del processo (core dump)



# Segnali di Unix

- **SIGFPE**
  - Generato quando si divide per 0
  - Default: terminazione e file core
- **SIGHUP**
  - Generato quando il terminale di controllo viene chiuso
  - Default: terminazione (quando si esce dalla shell tutti i processi in background ricevono un SIGHUP e terminano)
- **SIGINT**
  - Generato quando premiamo CTRL-C
  - Default: terminazione

# Segnali di Unix

- **SIGKILL**
  - Può essere mandato dal proprietario o da root
  - Non può essere ignorato, il processo termina
- **SIGSEGV**
  - Generato quando il processo tenta di accedere a memoria al di fuori del suo segmento
  - default: terminazione e file core
- **SIGUSR1, SIGUSR2**
  - Sono definiti dall'utente
  - default: terminazione

# La funzione `signal()`

- La funzione `signal()` fornisce lo strumento per istruire il kernel ad eseguire una determinata azione quando il processo chiamante riceve 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 quando si presenta tale segnale
- La funzione ritorna le **disposizioni** precedenti per il segnale

# La funzione `signal()` (cont.)

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

```
void (*signal(int signo, void (*func)(int)))(int);
```

- La funzione `signal()` ha due argomenti e ritorna un puntatore ad una funzione che non ritorna nulla (`void`)
  - `signo` è un intero (il nome del segnale occorso)
  - `func` è il puntatore ad una funzione che prende come argomento un intero e non ritorna nulla
  - Il valore di `func` è
    - La costante `SIG_IGN` (indica al sistema di ignorare il segnale)
    - La costante `SIG_DFL` (indica al sistema di associare l'azione di default)
    - L'indirizzo di una funzione da chiamare quando occorre il segnale (gestore del segnale)
- La funzione di cui è ritornato l'indirizzo come valore della funzione `signal` ha un unico argomento intero (l'ultimo `int`)

# Funzione signal()

- Il prototipo della funzione `signal` può essere reso più semplice mediante la seguente typedef

```
typedef void (*sighandler_t) (int);
```

Il prototipo diventa

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

- Esaminando l'header di sistema `<signal.h>`, troveremo (probabilmente) dichiarazioni della forma

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

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

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

- Queste costanti possono essere usate in luogo del secondo argomento di `signal` e del valore di ritorno da `signal`

# Esempio

```
#include <signal.h>

int main(void)
{
    signal (SIGINT, SIG_IGN);
    while (1);
}
```

Non è possibile interrompere l'esecuzione con <CTRL C>.

E' necessario eseguire un kill da un'altra shell!

# Esempio: Gestore che “cattura” vari segnali 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();
}
```

# Esempio: Gestore che “cattura” vari segnali e ne stampa il numero (cont.)

```
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;
}
```



# Esempio in esecuzione

```
$ ./a.out &  
[1] 774  
$ kill -USR1 774  
$ received SIGUSR1  
  
$ kill -USR2 774  
$ received SIGUSR2  
  
$ kill 774  
$ jobs  
[1]+  Terminated a.out  
$ ./a.out  
received SIGINT  
received SIGTSTP
```

La seconda volta vengono digitati <Ctrl C> e <Ctrl Z>.

Per terminare la seconda esecuzione è adesso necessario eseguire un kill(1) da un'altra shell!

# 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 possano fare di tutto, essi dovrebbero essere il più semplice possibile per evitare race condition
- Esempio:
  - Il gestore imposta una variabile globale (flag) ed esce. Il programma principale controlla ciclicamente il flag e se è impostato avvia le azioni corrispondenti
  - Il gestore esegue un qualche tipo di pulizia prima di terminare il processo

# 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 un processo, non può essere intercettato da un eseguibile avviato con `exec`, perché per quest'ultimo l'indirizzo della funzione di gestione del segnale non ha senso

# System call `kill()` e `raise()`

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

- La chiamata di sistema `kill()` permette ad un processo di inviare il segnale `signo` al processo `pid`, `raise()` a sé stesso
- Ritorna 0, se ok, <0 in caso di errore
- La chiamata a `raise (signo)` è equivalente alla chiamata `kill (getpid(), signo)`

# System call `kill()` e funzione `raise()`

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

- `pid` può assumere i seguenti valori:
- `pid > 0` il segnale viene inviato al processo con process ID uguale a `pid`
- `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`

# System call kill()

- POSIX.1 definisce il segnale numero 0 come **segnale nullo**
- Se l'argomento **signo** è 0, viene eseguito da **kill** il controllo normale dell'errore, ma non viene inviato alcun segnale
  - Usato per determinare se uno specifico processo esiste
  - Inviando un segnale nullo ad un processo che non esiste, **kill** restituirà -1
- 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

# Richiedere un segnale di allarme: alarm()

```
#include <unistd.h>
```

```
unsigned int alarm (unsigned int seconds)
```

- Istruisce il kernel a spedire il segnale **SIGALRM** al processo invocante dopo **seconds** secondi
- Un eventuale **alarm()** già schedulato viene sovrascritto col nuovo
- Se **seconds** è 0, non schedula nessun nuovo **alarm()** (e cancella quello eventualmente già schedulato)
- **Restituisce** il numero di secondi rimanenti prima dell'invio dell'allarme schedulato in precedenza, oppure 0 se non è schedulato nessun **alarm()**
- Nota: l'allarme è inviato dopo almeno **seconds** secondi, ma il meccanismo di scheduling può ritardare ulteriormente la ricezione del segnale

# Attendere un segnale: pause()

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

- Sospende il processo chiamante finché non è intercettato un segnale
- Pause restituisce il controllo solo dopo che è eseguito un gestore di segnale e, a sua volta, restituisce il controllo
  - Restituisce -1 e pone `errno = EINTR`



# Funzione sleep

```
#include <unistd.h>
```

```
unsigned int sleep(unsigned int seconds)
```

- Sospende il processo chiamante fino a che
  - trascorre la quantità di tempo specificata da **seconds**
  - il processo intercetta un segnale e il corrispondente gestore del segnale ritorna
- Come per **alarm**, il momento reale della restituzione del controllo da parte di **sleep** può essere qualche istante dopo rispetto alla richiesta a causa di altre attività del sistema

# Funzione nanosleep

```
#include <unistd.h>
```

```
int nanosleep(const struct timespec *req,  
struct timespec *rem)
```

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

# Funzione abort()

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

- Invia il segnale **SIGABRT** al chiamante (i processi non dovrebbero ignorare questo segnale)
- **POSIX.1** specifica che **abort** annulla l'eventuale blocco o ignoro del segnale dal processo
- Permettere ad un processo di intercettare **SIGABRT** equivale a consentirgli l'esecuzione di operazioni di pulizia prima della terminazione
- Se il processo non termina se stesso nel gestore del segnale, **POSIX.1** impone che quando il gestore del segnale ritorna, **abort** termina il processo

# Segnali inaffidabili

- Nelle prime versioni di UNIX i segnali erano inaffidabili
  - I segnali potevano andar 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

# Segnali inaffidabili: problemi (1)

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

```
int sig_int(); // funzione di gestione del segnale
```

```
...
```

```
signal(SIGINT, sig_int); // definizione del gestore
```

```
...
```

```
sig_int()
```

```
{
```

```
    signal(SIGINT, sig_int); // ridefinizione gestore
```

```
    ...
```

```
        // elaborazione segnale
```

```
}
```

# Segnali inaffidabili: problemi (2)

```
int sig_int_flag;
main() {
    int sig_int();
    ...
    signal(SIGINT, sig_int);
    ...
    while(sig_int_flag == 0)
        pause();
    ...
}
sig_int() {
    signal(SIGINT, sig_int);
    sig_int_flag=1;
}
```

- Il processo non ha la possibilità di bloccare il segnale quando si verifica
  - Al più può ignorarlo
- Ci sono casi in cui l'occorrenza di un segnale è necessario
  - prevenirla
  - ma ricordarla!

# Rientranza delle funzioni

- Non tutte le chiamate di sistema e le funzioni di libreria possono essere invocate con sicurezza da un gestore di segnale
- Per comprendere questo aspetto dobbiamo introdurre il concetto di **rientranza** delle funzioni
  - Per spiegare una funzione rientrante dobbiamo capire la differenza tra programmi a singolo thread e multithread
  - I classici programmi UNIX hanno un singolo thread di esecuzione: la CPU elabora le istruzioni per un singolo flusso logico di esecuzione attraverso il programma
  - In un programma multithread, ci sono più flussi logici indipendenti e concorrenti di esecuzione all'interno dello stesso processo
  - Il concetto di thread multipli di esecuzione è rilevante per programmi che impiegano i gestori dei segnali

# Rientranza delle funzioni (cont.)

- Un gestore può interrompere in modo asincrono (non predicibile) l'esecuzione di un programma in un qualsiasi istante temporale
  - il programma principale ed il gestore è come se costituissero due thread di esecuzione indipendenti (sebbene non concorrenti) all'interno dello stesso processo
- Una funzione è detta rientrante se può essere eseguita simultaneamente in modo **sicuro** da più thread di esecuzione nello stesso processo
  - Sicuro significa che che la funzione ottiene i risultati previsti indipendentemente dallo stato di esecuzione di qualsiasi altro thread



# Rientranza delle funzioni

- Un **funzione rientrante** è una funzione che può essere usata da più di un processo (o thread) in modo concorrente senza corrompere i dati
  - Una funzione rientrante può essere interrotta in qualsiasi momento e riavviata in un momento successivo senza la perdita di dati
    - Le funzioni rientranti usano **variabili locali** o **proteggono i propri dati** quando sono utilizzate le variabili globali
- Una **funzione non rientrante** è una funzione che non può essere condivisa da più task a meno che la mutua esclusione alla funzione sia assicurata usando un semaforo o disabilitando le interruzioni durante le sezioni critiche di codice

# Rientranza delle funzioni (cont.)

- Una funzione rientrante:
  - Non mantiene dati statici in chiamate successive
  - Non restituisce un puntatore a dati statici; tutti i dati sono passati da chi invoca la funzione
  - Usa dati locali o assicura protezione ai dati globali facendone una copia locale
  - Non deve chiamare alcuna funzione non-rientrante
- In generale, le funzioni non rientranti
  - Invocano malloc o free
  - E' noto che usano strutture dati statiche
  - Sono parte della libreria di I/O standard

# Segnali e rientranza

- Esempio
  - supponiamo che una funzione riceva un segnale mentre sta allocando della memoria con `malloc` e che la funzione di gestione del segnale a sua volta allochi memoria dinamicamente
  - Poiché `malloc` mantiene una lista di tutte le aree allocate, è possibile che il segnale venga lanciato mentre essa sta aggiornando tale lista e l'esecuzione del gestore del segnale intervenga proprio su tale lista. In tal caso il risultato è imprevedibile

# Segnali e rientranza (2)

- Un segnale può essere consegnato tra l'inizio e la fine di un operatore C che richiede più istruzioni
  - A livello del programmatore, l'istruzione può apparire atomica, sebbene possa, in realtà, richiedere più di un'istruzione macchina per completare l'operazione
    - Esempio
    - L'istruzione `temp+=1;`
      - Su un processore x86 potrebbe essere compilata come

```
mov ax, [temp]
inc ax
mov [temp], ax
```

# Libreria I/O standard e rientranza

- I problemi si verificano anche se si usano gli stream di I/O
- Supponiamo che il gestore stampa un messaggio con `printf()` ed il programma era nel mezzo di una chiamata a `printf()` usando lo stesso stream quando il segnale è stato consegnato
  - Entrambi il messaggio del gestore e i dati del programma potrebbero essere corrotti, poiché entrambe le chiamate operano sulla stessa struttura dati: lo stream stesso

# Funzioni rientranti (openBSD 3.0)

- **exit(), access(), alarm(), cfgetispeed(), cfgetospeed(), cfsetispeed(), cfsetospeed(), chdir(), chmod(), chown(), close(), creat(), dup(), dup2(), execl(), execve(), fcntl(), fork(), fpathconf(), fstat(), fsync(), getegid(), geteuid(), getgid(), getgroups(), getpgrp(), getpid(), getppid(), getuid(), kill(), link(), lseek(), mkdir(), mkfifo(), open(), pathconf(), pause(), pipe(), raise(), read(), rename(), rmdir(), setgid(), setpgid(), setsid(), setuid(), sigaction(), sigaddset(), sigdelset(), sigemptyset(), sigfillset(), sigismember(), signal(), sigpending(), sigprocmask(), sigsuspend(), sleep(), stat(), sysconf(), tcdrain(), tcflow(), tcflush(), tcgetattr(), tcgetpgrp(), tcsendbreak(), tcsetattr(), tcsetpgrp(), time(), times(), umask(), uname(), unlink(), utime(), wait(), waitpid(), write()**

```
#include<apue.h> // Esempio d'uso dei segnali
static void handler(int);
int main(){
    int i,n;
    n = fork();
    if (n == -1){
        fprintf(stderr,"fork fallita\n");
        exit(1);
    }
    else if ( n == 0)  /* processo figlio */
    {
        printf("\n(figlio) il mio process-id e` %d\n",getpid());
        printf("\n(figlio) aspetto 3 secondi e invio un segnale a %d\n",getppid());
        for (i=0; i<3; i++){
            sleep(1);
            printf(".\n");
        }

        kill(getppid(),SIGUSR1);
        printf("\n(figlio) ho finito e muoio\n");

        exit(0);
    }
}
```

```

else{          /* processo padre */
    signal(SIGUSR1,handler);
    printf("\n(padre) il mio process-id %d\n",getpid());
    printf("\n(padre) incomincio le mie operazioni\n");
    for (i = 0; i < 6; i++){
        sleep(1);
        printf("*\n");
    }
    printf("\n(padre) ora muoio anch'io\n");
    exit(0);
}
}

```

```

static void handler(int signo)
/* operazione alla ricezione di una kill */
{
    if (signo == SIGUSR1)
        printf("\n (padre) RICEVUTO INTERRUPT \n");
    return;
}

```



# Esercizio 1

1. Scrivere un programma che intercetta CTRL-c ed invece di terminare scrive su stdout “non voglio terminare”
2. Scrivere un programma C che riceva in input da tastiera due numeri interi, a e b, e ne stampi a video:
  - la somma "a+b" solo quando riceve il segnale SIGUSR2
  - la differenza "a-b" quando riceve il segnale SIGUSR1
  - Il programma esce quando riceve SIGINT

# Altri esempi

- Il programma **critical.c** suggerisce come si possono proteggere pezzi di codice “critici” da interruzioni dovute a **<Ctrl-c>** (segnale **SIGINT**) o da altri segnali simili che possono essere ignorati
- 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

# Esempio: critical.c

```
#include <apue.h>
int main (void) {
    void (*oldHandler) (int); /* vecchio handler*/
    printf ("Posso essere interrotto\n");
    sleep (3);
    oldHandler = signal (SIGINT, SIG_IGN); // Ignora
    Ctrl-c
    printf ("Ora sono protetto da Control-C\n");
    sleep (3);
    signal (SIGINT, oldHandler); /* Ripristina il
    vecchio handler */
    printf ("Posso essere interrotto nuovamente\n");
    sleep (3);
    printf ("Ciao!\n");
    return 0;
}
```

# Esempio: critical.c

```
$ ./critical
```

```
Posso essere interrotto
```

```
Ora sono protetto da Control-C
```

```
Posso essere interrotto nuovamente
```

```
Ciao!
```

```
$
```

# Esempio: limit.c (terminazione figlio)

- Il programma **limit.c** permette all'utente di limitare il tempo impiegato da un comando per l'esecuzione
- **limit nsec cmd args**
  - esegue il comando **cmd** con gli argomenti **args** indicati, dedicandovi al massimo **nsec** secondi
  - Il programma definisce un **handler** per il segnale **SIGCHLD**

# Esempio: limit .c (terminazione figlio)

```
#include <apue.h>
#include <sys/types.h>
#include <sys/wait.h>

int delay;
void childHandler (int); /* gestore terminazione figlio */

int main (int argc, char *argv[]) {
    int pid;
    signal (SIGCHLD, childHandler); /* Installa il gestore della
    figlio */
    pid = fork (); /* duplicazione */

    if (pid == 0) { /* Figlio */
        execvp (argv[2], &argv[2]); /* Esegue il comando */
    }
    else { /* Padre */
        sscanf (argv[1], "%d", &delay); /* delay da riga di comando */
        sleep (delay); /* dorme per delay secondi */
        printf ("Child %d exceeded limit and is being killed\n", pid);
        kill (pid, SIGINT); /* Kill il figlio */
    }

    return 0;
}
```

terminazione

# Esempio: limit.c (terminazione figlio)

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

```
$ limit 3 find / -name filechenonce
```

```
find: /root/.links: Permission denied
```

```
find: /root/.ssh: Permission denied
```

```
...
```

```
Child 828 exceeded limit and is being killed
```

```
$ limit 1 ls
```

```
count.c myexec.c redirect.c critical.c
```

```
limit.c limit myfork.c lez7.ps
```

```
Child 828 terminated within 1 seconds
```

```
$
```

## Esempio: pulse.c (sospensione&ripresa)

- Il programma **pulse.c** crea due figli che entrano in un ciclo infinito e mostrano un messaggio ogni secondo
- Il padre aspetta 2 secondi e quindi sospende il primo figlio, mentre il secondo figlio continua l'esecuzione
- Dopo altri 2 secondi il padre riattiva il primo figlio, aspetta altri 2 secondi e quindi termina entrambi i figli



# Esempio: pulse.c (sospensione&ripresa)

```
#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);
        }
    }
}
```

# Esempio: pulse.c (cont.)

```
/* ... continua ... */
sleep (2);
kill (pid1, SIGSTOP); /* Sospende il primo figlio */
sleep (2);
kill (pid1, SIGCONT); /* Ripristina il primo figlio */
sleep (2);
kill (pid1, SIGINT); /* termina il primo figlio */
kill (pid2, SIGINT); /* termina il secondo figlio */
return 0;
}
```

# Esempio: pulse.c (cont.)

*... funzionamento ...*

```
$ pulse  
pid1 is alive  
pid2 is alive  
pid1 is alive  
pid2 is alive  
pid2 is alive  
pid2 is alive  
pid1 is alive  
pid2 is alive  
pid1 is alive  
pid2 is alive  
pid1 is alive  
pid2 is alive
```

# Esercizio 2

- Scrivere un programma in C e Posix sotto Linux che, preso un argomento intero positivo da riga di comando, gestisca la seguente situazione:
  - genera due figli A e B e
    - se l'argomento è PARI invia un segnale SIGUSR1 alla ricezione del quale il figlio A calcola il cubo del numero passato come argomento da linea di comando, mentre il figlio B stampa un messaggio di arrivederci e termina.
    - se l'argomento è DISPARI invia un segnale SIGUSR2 alla ricezione del quale il figlio B calcola il reciproco del numero passato come argomento, attende per un numero di secondi pari al doppio del numero passato come argomento ed invia un segnale SIGUSR1 al processo A dopodiché termina l'esecuzione. Il figlio A, invece, attende la ricezione del segnale SIGUSR1, stampa un messaggio e termina.

# Segnali affidabili

- Uno dei problemi più insidiosi che può verificarsi quando si gestisce un segnale è l'occorrenza di un secondo segnale mentre è in esecuzione la funzione di gestione del segnale
  - Il secondo segnale può essere di un tipo diverso rispetto a quello correntemente in gestione, od anche dello stesso tipo
- E' necessario prendere delle precauzioni all'interno della funzione di gestione del segnale per evitare race
- Unix contiene alcune caratteristiche che consentono di bloccare l'eventuale elaborazione dei segnali

# Maschera dei segnali

- La chiamata di sistema POSIX usata per mascherare i segnali è `sigprocmask()`
- Consente di specificare un insieme di segnali da bloccare, e restituisce la lista dei segnali che erano bloccati precedentemente
  - Utile per ripristinare lo stato della maschera precedente, una volta completata la nostra sezione critica

# sigprocmask()

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

```
int sigprocmask(int how, const sigset_t *set,  
sigset_t *oset);
```

- **int how**
  - Definisce se aggiungere segnali alla maschera corrente del processo (**SIG\_BLOCK**), rimuoverli dalla maschera corrente (**SIG\_UNBLOCK**) o sostituire completamente la maschera corrente con una nuova (**SIG\_SETMASK**)
- **const sigset\_t \*set**
  - L'insieme dei segnali da bloccare, aggiungere o rimuovere dalla maschera corrente sulla base del parametro **how**
- **sigset\_t \*oset**
  - Se non è NULL, conterrà la maschera precedente
  - Dopo aver invocato sigprocmask, se qualche segnale non bloccato è pendente, almeno uno di tali segnali sarà consegnato al processo prima che sigprocmask ritorni

# Il tipo di dato `sigset_t`

- Tipicamente i segnali possono essere messi in corrispondenza con ciascun bit di un intero
- Il numero di segnali differenti che possono occorrere può superare il numero di bit in un intero
  - In generale, potremmo non usare un intero per rappresentare l'insieme con un bit per segnale
  - POSIX definisce il tipo di dato `sigset_t` per contenere un insieme di segnali ed un insieme di funzioni per manipolarlo



# Insiemi di segnali

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

```
int sigemptyset(sigset_t *set);
```

```
int sigfillset(sigset_t *set);
```

```
int sigaddset(sigset_t *set, int signo);
```

```
int sigdelset(sigset_t *set, int signo);
```

Ritornano 0 se OK, -1 in caso di errore

```
int sigismember(const sigset_t *set, int signo);
```

Ritorna 1 se vero, 0 se falso, -1 in caso di errore

# Esempi

```
/* definisce una nuova maschera */
sigset_t mask_set;
/* svuota la maschera */
sigemptyset(&mask_set);
/* aggiunge i segnali TSTP e INT alla maschera */
sigaddset(&mask_set, SIGTSTP);
sigaddset(&mask_set, SIGINT);
/* rimuove il segnale TSTP dall'insieme */
sigdelset(&mask_set, SIGTSTP);
/* controlla se il segnale INT è definito nell'insieme */
if (sigismember(&mask_set, SIGINT)
    printf("segnale INT nell'insieme\n");
else
    printf("segnale INT non nell'insieme\n");
/* pone tutti i segnali del sistema nell'insieme */
sigfillset(&mask_set)
```

# Esempio

- Vediamo un breve esempio (pezzi) di codice che conta il numero di segnali Ctrl-C digitati dall'utente
  - la quinta volta chiede all'utente se vuole terminare
  - se l'utente digita Ctrl-Z è stampato il numero di Ctrl-C digitati

```
/* Definiamo il contatore di Ctrl-C counter inizializzato a 0 */
int ctrl_c_count = 0;
#define CTRL_C_THRESHOLD 5
void catch_int(int sig_num) { /* gestore segnale Ctrl-C */
    sigset_t mask_set; /* per impostare la maschera dei segnali */
    sigset_t old_set; /* memorizza la vecchia maschera */

    /* maschera gli altri segnali mentre siamo nel gestore*/
    sigfillset(&mask_set);
    sigprocmask(SIG_SETMASK, &mask_set, &old_set);
```

# Esempio (cont.)

```
ctrl_c_count++; /* incrementa count, e controlla la soglia */

if (ctrl_c_count >= CTRL_C_THRESHOLD) {
    char answer[30];
    printf("\nVuoi terminare? [s/N]: ");
    fflush(stdout);
    fgets(answer, 20, stdin);

    if (answer[0] == 's' || answer[0] == 'S') {
        printf("\nTermino...\n");
        fflush(stdout);
        exit(0);
    }
    else {
        printf("\nContinuo\n");
        fflush(stdout);
        /* reset del contatore per Ctrl-C */
        ctrl_c_count = 0;
    }
}

signal(SIGINT, catch_int);
}
```

# Esempio (cont.)

```
void catch_suspend(int sig_num) {
    sigset_t mask_set; /* usato per mascherare i segnali */
    sigset_t old_set; /* memorizza la vecchia maschera */
    sigfillset(&mask_set);
    sigprocmask(SIG_SETMASK, &mask_set, &old_set);
    /* stampa il contatore corrente per Ctrl-C */
    printf("\n\n'%d' Ctrl-C digitazioni\n", ctrl_c_count);
    fflush(stdout);
    signal(SIGTSTP, catch_suspend);
}
/* da qualche parte nel main... */ . .
/* imposta i gestori per il Ctrl-C e Ctrl-Z */
signal(SIGINT, catch_int);
signal(SIGTSTP, catch_suspend);

. . /* ed il resto del programma */ . .
```

# Evitare le race condition

- L'uso di `sigprocmask()` nell'esempio non risolve tutte le possibili race condition
  - Ad esempio, dopo l'entrata nel gestore del segnale, ma prima di invocare `sigprocmask()`, è possibile ricevere un altro segnale, che sarà gestito
  - Pertanto, se l'utente è molto veloce (o il sistema molto lento), si crea una race
- Il modo per evitare completamente race, è consentire al sistema di impostare la maschera dei segnali prima che sia chiamato il gestore del segnale
  - Ciò può essere fatto invocando la system call `sigaction()` per definire sia la funzione di gestione del segnale che la maschera dei segnali da usare quando è eseguito il gestore

# sigaction

- Consente di esaminare o modificare (o entrambe le cose) l'azione associata con un particolare segnale
  - Questa funzione sostituisce la funzione **signal** nelle versioni più recenti di Unix

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

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

- **signo** è il numero del segnale la cui azione stiamo esaminando o modificando
- se il puntatore **act** è non nullo, stiamo modificando l'azione
- se **oact** è non nullo, il sistema ritorna l'azione precedente per il segnale attraverso il puntatore **oact**

# La struttura sigaction

```
struct sigaction {  
    void (*sa_handler) (int);  
    sigset_t sa_mask;  
    int sa_flags;  
    void (*sa_sigaction) (int, siginfo_t *, void *);  
};
```

- Quando si cambia l'azione per un segnale, se il campo **sa\_handler** contiene l'indirizzo di una funzione per la sua gestione (e no SIG\_IGN o SIG\_DFL)
  - Il campo **sa\_mask** specifica l'insieme dei segnali che sono aggiunti alla maschera dei segnali prima che la funzione di gestione sia chiamata
  - Se e quando la funzione di gestione del segnale ritorna, la maschera dei segnali del processo è resettata al suo valore precedente
  - Il SO include il segnale attualmente in consegna nella maschera dei segnali quando il gestore è invocato
    - In questo modo siamo sicuri che mentre stiamo elaborando un dato segnale, un'altra occorrenza di quel segnale sarà bloccata fino a che non abbiamo completato la gestione del precedente



# Interruzione e riavvio di system call

- Consideriamo il seguente scenario
  - Definiamo un handler per un dato segnale
  - Facciamo una chiamata di sistema bloccante, ad esempio, una read da un terminale che si blocca fino a che non è fornito un input
  - Mentre la system call è bloccata, il segnale per cui abbiamo definito il gestore è consegnato ed è invocato il suo gestore
- Cosa accade dopo che il gestore ritorna?
  - Di default, la system call fallisce con un errore EINTR (“Interrupted function”). Questa caratteristica può essere utile
  - Spesso però preferiamo continuare l’esecuzione di una system call interrotta. Per fare ciò, possiamo usare del codice come il seguente per riavviare normalmente una system call nel caso fosse interrotta da un gestore di segnale:

# Flag SA\_RESTART

```
while (( cnt = read(fd,buf, BUF_SIZE)) ==-1 && errno == EINTR)
    Continue;

if (cnt == -1)
    errExit("read");
```

Anche con un tale codice, avere gestori che interrompono le system call può non essere conveniente, poiché dobbiamo aggiungere il codice ad ogni system call bloccante (assumendo che noi vogliamo riavviarla)

Possiamo usare il flag SA\_RESTART in modo che le system call sono automaticamente riavviate dal kernel per conto del processo. Ciò significa che non dobbiamo gestire un possibile errore EINTR ritornato da queste system call

# Segnali per il controllo dei job

# Gruppi di processi

- Ogni processo è membro di un gruppo di processi
- Un gruppo di processi ha associato un identificatore `pgid` – process group id (da non confondere con il gid)
- Un processo figlio eredita il gruppo di appartenenza dal padre
- La system call `setpgid()` permette al processo invocante di cambiare gruppo di appartenenza (a se stesso o ad altri)

# Gruppi di processi e terminale di controllo

- Ad ogni processo può essere associato un **terminale di controllo**: è tipicamente il terminale da cui il processo è lanciato
  - i figli ereditano il terminale di controllo del padre
  - se un processo esegue una `exec()`, il terminale di controllo non cambia
- Ad ogni terminale è associato un **processo di controllo**
  - Se il terminale individua un metacarattere, come `<Ctrl-C>`, spedisce il segnale appropriato a tutti i processi nel gruppo del processo di controllo
  - se un processo tenta di leggere dal suo terminale di controllo e non è membro del gruppo del processo di controllo di quel terminale, il processo riceve un segnale `SIGTTIN` che, normalmente, lo sospende (`SIGTTOU` se tenta di scrivere sul terminale di controllo)

# Gruppi e terminale: uso nella shell

- All'avvio di una shell interattiva
  - la shell è il processo di controllo del terminale da cui è lanciata
- Se la shell esegue un **comando in foreground**
  - la shell figlio si mette in un diverso gruppo, assume il controllo del terminale, esegue il comando
  - così ogni segnale generato dal terminale viene indirizzato al comando e non alla shell originaria
  - quando il comando termina, la shell originaria riprende il controllo del terminale
- Se la shell esegue un **comando in background**
  - la shell figlio si mette in un diverso gruppo ed esegue il comando, ma non assume il controllo del terminale
  - così ogni segnale generato dal terminale continua ad essere indirizzato alla shell originaria
  - se il comando in background tenta di leggere dal suo terminale di controllo, viene sospeso da un segnale SIGTTIN

# Cambiare il gruppo: setpgid()

```
int setpgid(pid_t pid, pid_t pgrpId)
```

- assegna valore **pgrpId** al process group id del processo con **PID pid**
  - se **pid** è **0**, cambia il valore del process group id del processo invocante
  - se **pgrpId** è **0**, assegna il process group ID del processo con **PID pid** al processo invocante
- Ha successo se
  - il processo invocante ed il processo specificato come primo argomento hanno lo stesso (effective) uid/gid
  - il proprietario del processo invocante è il superuser
- Restituisce 0 se ha successo; -1 altrimenti

# Cambiare il gruppo: setpgid()

- Quando un processo vuole creare un proprio gruppo di processi, distinto dagli altri gruppi del sistema, tipicamente passa il proprio **PID** come argomento per **setpgid()**

```
setpgid(0, getpid())
```



# Ottenere il gruppo: getpgid()

```
pid_t getpgid(pid_t pid)
```

- Restituisce il gruppo di processi a cui appartiene il processo con **PID pid**
- Se pid è 0, restituisce il gruppo di processi a cui appartiene il processo invocante
- Non fallisce mai

# Esempio: pgrp1.c

- Il programma **pgrp1.c** mostra come un terminale invia i segnali ad ogni processo appartenente al gruppo di processi del suo processo di controllo
- Poiché un figlio eredita il gruppo di processi del padre, padre e figlio intercettano il segnale **SIGINT**

# Esempio: pgrp1.c

```
#include <signal.h>
#include <stdio.h>
void sigHandler (int sig) {
    printf ("Process %d got a %d signal\n", getpid(), sig);
}

int main (void) {
    signal (SIGINT, sigHandler); /* Gestisce Control-C */

    if (fork () == 0)
        printf ("Child PID %d PGRP %d waits\n",getpid(),getpgid(0));
    else
        printf ("Parent PID %d PGRP %d waits\n",getpid(),getpgid(0));

    pause (); /* Aspetta un segnale */
}
```

# Esempio: pgrp1.c

```
$ ./pgrp1
```

```
Parent PID 24444 PGRP 24444 waits
```

```
Child PID 24445 PGRP 24444 waits
```

```
<^C> Process 24445 got a 2 signal
```

```
Process 24444 got a 2 signal
```

```
$
```

# Esempio: pgrp2.c

- Il programma pgrp2.c mostra che se un processo lascia il gruppo del processo di controllo del terminale, non riceve più segnali dal terminale

# Esempio: pgrp2.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
void sigHandler (int sig) {
    printf ("Process %d got a SIGINT\n", getpid());
    exit (1);
}

int main (void) {
    int i;
    signal (SIGINT, sigHandler); /* gestore di segnali */

    if (fork () == 0)
        setpgid (0, getpid ()); /* il figlio nel suo gruppo */

    printf("Process PID %d PGRP %d waits\n",getpid(),getpgid(0));

    for (i = 1; i <= 3; i++) { /* Cicla 3 volte */
        printf ("Process %d is alive\n", getpid());
        sleep(2);
    }

    return 0;
}
```

# Esempio: pgrp2.c

- esecuzione...

```
$ ./pgrp2
```

```
Process PID 24535 PGRP 24535 waits
```

```
Process 24535 is alive
```

```
Process PID 24536 PGRP 24536 waits
```

```
Process 24536 is alive
```

```
<^C> Process 24535 got a SIGINT
```

```
$ Process 24536 is alive
```

```
Process 24536 is alive
```

```
<return>
```

```
$
```