# Terza Esercitazione

Gestione dei segnali in Unix
Primitive signal e kill

Andrea Reale
andrea.reale@unibo.it

# Agenda

- **Esercizio 1**
  - Primi passi con la gestione dei segnali in Unix
- **Esercizio 2**
  - Contenuto informativo dei segnali Unix
- **Esercizio 3**
  - Sincronizzazione tra processi tramite segnali

# Tabella Riassuntiva

| | |
|---|---|
| **signal** | • **Imposta l'azione** da eseguire in risposta alla ricezione di un segnale (può essere anche SIG_IGN o SIG_DFL)<br>• Il comportamento può variare a seconda della versione di Unix utilizzata (vedi man 2 signal)<br>• **kill -l** da una shell per una lista dei segnali disponibili |
| **kill** | • **Invio di un segnale ad un processo**<br>• Va specificato sia il segnale che il processo destinatario<br>• Restituisce 0 se tutto va bene o -1 in caso di errore |
| pause | • Chiamata **bloccante:** il processo si sospende fino alla ricezione di un qualisiasi segnale<br>• Restituisce sempre -1 (se il processo non viene terminato dal segnale) |
| alarm | • "Schedula" l'invio del segnale **SIGALRM** al processo chiamante dopo un ritardo specificato come argomento |
| sleep | • **Sospende** il processo chiamante per un numero intero di secondi, **oppure fino all'arrivo di un segnale**<br>• Restituisce il numero di secondi che sarebbero rimasti da dormire (0 se nessun segnale è arrivato) |

# **signal** in Linux (con glibc)

- Alla fine dell'esecuzione di un handler definito dall'utente, il sistema si occupa di reinstallarlo automaticamente
- Se, durante l'esecuzione di un handler, arriva un secondo segnale uguale a quello che ha causato la sua esecuzione, il segnale viene **bloccato** e gestito una volta terminato il primo handler.
- Segnali che arrivano mentre c'è un segnale uguale bloccato vengono "accorpati" in uno: **i.e.** solo un segnale verrà consegnato al processo
  - questo vale anche se molte chiamate a kill() vengono eseguite in tempi ravvicinati.

# Esercizio 1

Segnali e stato di terminazione

## Esercizio 1 - Traccia (1/2)

Si realizzi un programma C che utilizzi le primitive Unix per la gestione di processi e segnali, con la seguente interfaccia di invocazione

**scopri_terminazione N**

Specifiche:

- Il processo lanciato genera N figli
  - I primi K processi **attendono** la ricezione del sengale **SIGUSR1** da parte del padre, e poi terminano.
  - I rimanenti processi **attendono** 5 secondi e poi terminano.

**NOTA**: K è il più grande intero <= N/2

# Esercizio 1 - Traccia (2/2)

- Tutti i figli devono **gestire la loro terminazione**
  - Nel caso specifico **stampano a video il loro PID** prima di terminare
- Gestire appropriatamente l'**attesa** dei figli
  - **No attesa attiva**
  - Quali **primitive** usare nell'un caso e nell'altro?
- Il padre termina metà dei suoi figli tramite SIGUSR1
  - Come fa a discriminare a quali figli inviarlo?

# Esercizio 1 - Schema di soluzione (1/2)

```
int main(int argc, char* argv[]) {
   int i, n, pid[MAX_CHILDREN];
   n = atoi(argv[1]);
   for(i=0; i<n; i++) {
      pid[i] = fork();
      if ( pid[i] == 0 ) {
         if (i < n/2) {
            wait_for_signal();
         }
         else {
            sleep_and_terminate();
         }
      }
      else if ( pid[i] > 0 ) { /* Codice Padre */}
      else { /* Gestione errori */}
   }
   for (i=0; i<n/2; i++) {
      kill(pid[i], SIGUSR1);
   }
   for (i=0; i<n; i++) {
      wait_child();
   }

   return 0;
}
```

# Esercizio 1 - Schema di soluzione (2/2)

```
void wait_for_signal()
{
   void (*sighandler_ptr)(int);
   sighandler_ptr = &sig_usr1_handler;
   /* Imposto il gestore dei segnali di tipo SIGUSR1 */
   signal(SIGUSR1, sighandler_ptr);
   pause();
   exit(EXIT_SUCCESS);
}
void sleep_and_terminate()
{
   sleep(5); /* Cosa succede se arriva un segnale durante la sleep? */
   printf("%d: Finished waiting 5 second. Now exiting.\n",getpid());
   exit(EXIT_SUCCESS);
}

void sig_usr1_handler(int signo)
{
   /* Gestione del segnale */
   printf("%d: received SIGUSR1. Will terminate...\n", getpid());
}

void wait_child() {
   ...
   pid = wait(&status);
   /* Gestione condizioni di errore e verifica tipo di terminazione (volontaria o da segnale) */
   ...
}
```

# Esercizio 2

## Comunicare informazioni tramite segnali

# Esercizio 2 - Traccia

Si realizzi un programma C che, utilizzando le appropriate primitive Unix, esegua il seguente comportamento:

- Il processo padre P0 genera 3 figli
- Ciascun figlio, **attende** un numero casuale di secondi (da 0 a 5) **dopodiché** - se quel numero è pari - invia un segnale a P0
- P0 stampa a video il PID dei processi da cui ha ricevuto un segnale
- P0 termina una volta che tutti i suoi figlio sono terminati. (**ATTENZIONE**: può anche capitare che nessuno dei figli gli invii un segnale)

# Esercizio 2 - Criticità

- Quando P0 riceve un segnale, come distingue quale processo l'ha inviato?
  - Quanti e quali segnali sono lasciati all'uso del programmatore?

- P0 deve contemporaneamente aspettare la terminazione di **tutti** i figli e l'arrivo di eventuali segnali
  - Cosa succede se un segnale arriva mentre P0 è bloccato sulla `wait()`?

# Parentesi - Generazioni numeri casuali

```
#include <stdlib.h>
...
int rand_num, delay;
/* Imposta un seed dipendente dal processo*/
srand(getpid());

/* Genera il numero casuale */
rand_num = rand();
/* rand_num è nell'intervallo [0, RAND_MAX] */

/* Il resto della divisione per 6 dà un numero tra 1 e 5/*
delay = rand_num % 6;
...
```

# Schema di soluzione (1/3)

```
int main(int argc, char* argv[]) {
   ...
   signal(SIGUSR1, &signal_handler); /*Figlio 1*/
   signal(SIGUSR2, &signal_handler); /*Figlio 2*/
   /* Cosa succede se arriva SIGTERM, ad esempio, da shell? */
   signal(SIGTERM, &signal_handler); /*Figlio 3*/

   parent_pid = getpid();
   for ( i=0; i < 3; i++) {
      /* pid[3] variabile globale*/
      pid[i] = fork();
      if ( pid[i] == 0 ) { /* Figlio */
         codice_figlio(i, parent_pid); /*i dice al figlio che numero è*/
      }
      else if ( pid[i] < 0 ){
         perror("Impossibile fare la fork()");
         exit(2);
      }
   }
   for ( i=0; i<3 ; i++ ) {
      wait_child();
   }

   return 0;
}
```

# Schema di soluzione (2/3)

```
void codice_figlio(int num, int parent_pid)
{
   ...
   srand(getpid());
   rand_num = rand();
   delay = rand_num % 6;
   sleep(delay);
   if ( (delay % 2) == 0 ) {
      switch (num) {
         case 0: /*Primo figlio*/
            signo = SIGUSR1;
            break;
         case 1: /*Secondo figlio*/
            signo = SIGUSR2;
            break;
         case 2: /*Terzo figlio*/
            signo = SIGTERM;
            break;
         default:
            fprintf(stderr, "Numero di figlio inaspettato!\n");
            exit(3);
      }
      kill( parent_pid, signo);
   }
   exit(0);

}
```

# Schema di soluzione (3/3)

```
void signal_handler(int signo) {
   int source;
   switch (signo) {
      case SIGUSR1:
         source = pid[0];
         break;
      case SIGUSR2:
         source = pid[1];
         break;
      case SIGTERM:
         source = pid[2];
         break;
      default:
         fprintf(stderr, "Segnale inaspettato\n");
         exit(5);
   }
   printf("Ricevuto un segnale da PID %d\n", source);
   return;
}
```

# Esercizio 3

Sincronizzazione tramite segnali

## Esercizio 3 - Traccia (1/2)

Si realizzi un programma che, utilizzando le system call Unix appropriate, abbia interfaccia di invocazione:

**esegui_in_sequenza N COM1 COM2**

- N e` un intero positivo
- COM1 e COM2 sono stringhe che rappresentano il nome di un file eseguibile

Il programma deve soddisfare le seguenti specifiche:

# Esercizio 3 - Traccia (2/2)

- Il processo iniziale (P0) deve creare due processi figli (**fratelli**) P1 e P2
- I processi P1 e P2 devono eseguire rispettivamente i comandi COM1 e COM2, rispettando il segunte vincolo:
  - **COM1 va eseguito soltanto al termine dell'esecuzione con successo di COM2.**
- Ciascuno dei figli, in caso di fallimento della primitiva exec usata per eseguire COM1 o COM2, **dovra` notificare tale evento al padre P0**, e successivamente terminare.
- **Comunque vada**, dopo N secondi (N > 0) dalla sua creazione, il processo P1 dovra` **terminare la propria esecuzione**.

# Note alla soluzione

- P0 crea P1 e P2: fork() e **attende**
- P1 deve terminare entro e non oltre N secondi (**qualunque sia il programma che esegue**)
  - Impostazione del timeout (che **primitiva** si potrebbe usare?)
- In caso di fallimento nell'esecuzione di uno dei comandi, il figlio responsabile ne deve dare notifica a P0
  - Invio dei segnali e relativa gestione
- Gestione dell'esecuzione **sequenzializzata** (prima COM2, poi COM1)
  - P1 e P2 sono fratelli: quali strumenti a disposizione per sincronizzarli?
  - Puo` P2 eseguire COM2 e, in caso di successo, inviare un segnale a P1?

# Variante

- Il padre P0, mentre il figlio esegue, continua la propria attivita` (cioe` non si pone in attesa di P1 e P2), e stampa il suo PID ripetutamente.

- Tuttavia, non appena ognuno dei suoi figli terminera`, P0 dovra` tempestivamente raccogliere e stampare il suo stato di terminazione.

vedi gestione del segnale **SIGCHLD**