

# Terza Esercitazione

Gestione di segnali in Unix  
Primitive `signal` e `kill`

---

# Primitive fondamentali

signal	<ul style="list-style-type: none"><li>• Imposta la reazione del processo all'eventuale ricezione di un segnale (può essere una funzione handler, SIG_IGN o SIG_DFL)</li></ul>
kill	<ul style="list-style-type: none"><li>• Invio di un segnale ad un processo</li><li>• Va specificato sia il segnale che il processo destinatario</li><li>• Restituisce 0 se tutto va bene o -1 in caso di errore</li><li>• <code>kill -1</code> da shell per una lista dei segnali disponibili</li></ul>
pause	<ul style="list-style-type: none"><li>• Chiamata <b>bloccante</b>: il processo si sospende fino alla ricezione di un qualsiasi segnale</li></ul>
alarm	<ul style="list-style-type: none"><li>• "Schedula" l'invio del segnale <b>SIGALRM</b> al processo chiamante dopo un intervallo di tempo specificato come argomento</li></ul>
sleep	<ul style="list-style-type: none"><li>• <b>Sospende</b> il processo chiamante per un numero intero di secondi, oppure fino all'arrivo di un segnale</li><li>• Restituisce il numero di secondi che sarebbero rimasti da dormire (0 se nessun segnale è arrivato)</li></ul>

# signal in Linux

- **Persistenza dell'handler**: 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 (e.g. molte chiamate a `kill()` eseguite in tempi ravvicinati).
-

# Digressione: puntatori a funzione

- C permette di definire puntatori a funzione che memorizzano l'indirizzo di memoria di una funzione
- Dichiarazione:

```
void (*pfunc) (int) ;
```

Dichiara una variabile "puntatore a funzione" pfunc. Tale funzione deve restituire void e accettare un parametro intero

- Assegnamento:

```
pfunc = &nome_funzione;
```

```
(oppure: pfunc = nome_funzione;)
```

- Un puntatore a funzione può essere passato come parametro ad altre funzioni
  - la funzione a cui punta può essere invocata:  

```
(*pfunc) (a) ; /* a variabile intera */
```
-

# Signature di `signal`

- Da: **man `signal`**  

```
typedef void (*sighandler_t)(int);  
sighandler_t signal(int signum,  
                    sighandler_t handler);
```
- è una funzione che prende due argomenti:
  - Il primo è un intero (`signum`)
  - Il secondo è di tipo `sighandler_t`
- e restituisce:
  - un valore di tipo `sighandler_t`
- A sua volta, il tipo `sighandler_t` è un puntatore a funzione
  - Che accetta un intero come argomento
  - E restituisce `void`
- Signature alternativa:  

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

# Un buon riferimento per la gestione dei segnali

[http://www.gnu.org/software/libc/manual/html\\_node/Signal-Handling.html#Signal-Handling](http://www.gnu.org/software/libc/manual/html_node/Signal-Handling.html#Signal-Handling)

---

# Esempio1 - Segnali di stato e terminazione

- 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**

- Il processo lanciato genera **N figli**
    - I primi **K** processi **attendono** la ricezione del segnale **SIGUSR1** da parte del padre, e poi terminano.
    - I **rimanenti** processi **attendono** 5 secondi e poi terminano.
  - **NOTA:** **K** è il più grande intero  $\leq N/2$
-

# Esempio1 (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 per i due tipi di figli?
  - Il padre termina metà dei suoi figli tramite **SIGUSR1**
    - Come fa a discriminare a quali figli inviarlo?
-



# Esempio 1 - Soluzione (1/3)

```
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 ) { /* Codice Figlio*/
            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;
}
```

---

# Esempio 1 - Soluzione (2/3)

```
void wait_for_signal(){
    /* Imposto il gestore dei segnali di tipo SIGUSR1 */
    signal(SIGUSR1, sig_usr1_handler);
    pause();
    exit(EXIT_SUCCESS);}

void sleep_and_terminate() {
    sleep(5); /*se arriva un segnale durante la sleep?*/
    printf("%d: Finished waiting 5 second.\n",getpid());
    exit(EXIT_SUCCESS);}

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

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

---

# Esempio 1 - Riflessione A:

```
void wait_for_signal(){
    /* Imposto il gestore dei segnali di tipo SIGUSR1 */
    signal(SIGUSR1, sig_usr1_handler);
    pause();
    exit(EXIT_SUCCESS);}

void sleep_and_terminate() {
    sleep(5); /*se arriva un segnale durante la sleep?*/
    printf("%d: Finished waiting 5 second.\n",getpid());
    exit(EXIT_SUCCESS);}

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

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

Cosa succede se SIGUSR1 arriva qui?

# Esempio 1 - Riflessione A

- Se il segnale **SIGUSR1** inviato dal padre arriva prima che il figlio abbia dichiarato qual è l'handler deputato a riceverlo, (quindi prima di `signal(SIGUSR1, sig_usr1_handler);`), il figlio esegue l'handler di default del segnale **SIGUSR1**: `exit`. Incidentalmente il comportamento è simile a quanto ci era richiesto, ma non verrà eseguita la `printf` di `sig_usr1_handler`.
  - Si può evitare con certezza che ciò accada? **NO!**
  - Tutto ciò che posso fare è far dormire il padre per un po' prima di fargli inviare **SIGUSR1**, ma non ho alcuna certezza che questo risolva sempre il problema!
-


# Esempio 1 - Riflessione B:

```
void wait_for_signal(){
    /* Imposto il gestore dei segnali di tipo SIGUSR1 */
    signal(SIGUSR1, sig_usr1_handler);
    pause();
    exit(EXIT_SUCCESS);}

void sleep_and_terminate() {
    sleep(5); /*se arriva un segnale durante la sleep?*/
    printf("%d: Finished waiting 5 second.\n",getpid());
    exit(EXIT_SUCCESS);}

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

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



E se SIGUSR1 arriva qui!?

# Esempio 1 - Riflessione B

- Se il segnale **SIGUSR1** arriva dopo la dichiarazione dell'handler, ma prima della **pause()**?
- Il figlio riceve il segnale, esegue correttamente l'handler e si mette in attesa... di un segnale che è già arrivato!  
=> il figlio attende all'infinito!
- Si può evitare tutto ciò? **SI!**
- Mettendo la **exit** nell' handler:

```
void sig_usr1_handler(int signum){  
    printf("%d: received SIGUSR1(%d). Will  
           terminate...\n", getpid(), signum);  
    exit(EXIT_SUCCESS);  
}
```

---

# Esercizio 2

Si realizzi un programma che, utilizzando le system call Unix appropriate, abbia interfaccia d'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

---

# Esercizio 2

- Il processo iniziale (**P0**) deve creare **2 processi figli**  
    □ **P1 e P2**
  - I processi P1 e P2 devono eseguire rispettivamente i comandi COM1 e COM2, rispettando il seguente 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), dovrà **notificare tale evento al padre P0**, e successivamente terminare.
  - Comunque vada, dopo N secondi ( $N > 0$ ) dalla sua creazione, il processo P1 dovrà **terminare la propria esecuzione.**
-



# Note alla soluzione (1/2)

- P0 crea P1 e P2: `fork()` e **attende**
  - P1 deve terminare entro e non oltre N secondi (qualunque sia il programma che esegue)
    - => Occorre una primitiva che imposti un **timeout**!
  - In caso di **fallimento** nell'esecuzione di uno dei comandi, il figlio responsabile ne deve dare notifica a P0
    - => Occorre usare dei segnali e gestirli con un **handler**
-

# Note alla soluzione (2/2)

Esecuzione **sequenzializzata** (prima COM2, poi COM1)

- P1 e P2 sono fratelli: come sincronizzarli? **P2 deve conoscere il pid di P1 per mandargli un segnale!**
- P2 deve prima eseguire COM2 => `exec()`. Poi, se tutto va bene, inviare un segnale a P1.

**Ma...**

La `exec()` sostituisce codice e dati del processo chiamante:

```
execl("/usr/bin/gcc", "gcc", "prog.c", "-o", "prog", (char*)0);  
perror("Errore in execl\n");  
exit(1);
```

Può P2 eseguire COM2 e, in caso di successo, inviare un segnale a P1? ... Può farlo eseguire a qualcun'altro? Devo generare **ALMENO** P1 e P2, ma non sono obbligato a generare solo loro!

---

# Esercizio 3

Realizzare una variante dell' esercizio 2:

- Il padre P0, mentre il figlio esegue, continua la propria attività (cioè non si pone in attesa di P1 e P2), e stampa il suo PID ripetutamente.
- Tuttavia, non appena ognuno dei suoi figli terminerà, P0 dovrà tempestivamente raccogliere e stampare il suo stato di terminazione.

•vedi gestione del segnale **SIGCHLD**

---