

Terza Esercitazione

Gestione di segnali in Unix

Primitive **signal** e **kill**

Primitive fondamentali (sintesi)

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

Esempio – 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 K

- Il processo iniziale genera **N figli**:
 - I primi **K** ($K < N$) processi **attendono** la ricezione del segnale **SIGUSR1** da parte del padre, e poi terminano.
 - I **rimanenti** processi **attendono 5 secondi** e poi terminano.
 - **Tutti** i figli devono **stampare a video il proprio PID** prima di terminare
-

Esempio - osservazioni

- Gestire appropriatamente le **attese**:
 - **No attesa attiva**
 - Quali **primitive** usare per i due tipi di figli?
 - Il padre termina K figli tramite **SIGUSR1**
 - Come fa a discriminare a quali figli inviarlo?
-

Esempio - Soluzione (1/3)

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

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

```
void sleep_and_terminate() {
    sleep(5);
    printf("%d: Slept 5sec. Withdrowing.\n",getpid());
    exit(EXIT_SUCCESS);}
```

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

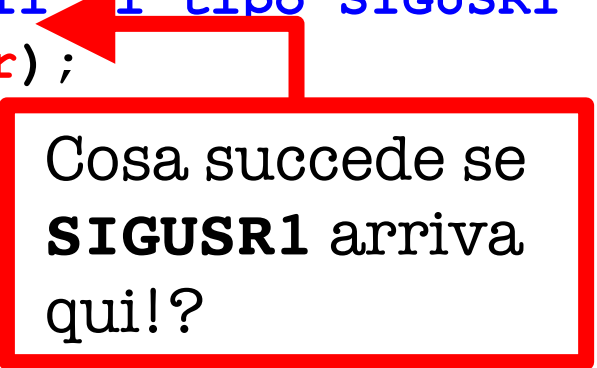
Esempio - Riflessione A

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

void sig_usr1_handler(int signum) {
    printf("%d: received SIGUSR1(%d)\n", getpid(), signum);
    terminate :-( \n", getpid(), signum);}

void sleep_and_terminate() {
    sleep(5);
    printf("%d: Slept 5sec. Withdrowing.\n",getpid());
    exit(EXIT_SUCCESS);}

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 - 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?
-

Esempio – Riflessione A

Soluzioni possibili:

- Far **dormire** il padre per un po' prima di fargli inviare **SIGUSR1** , ma non ho alcuna certezza che questo risolva sempre il problema!
 - Far eseguire la **signal(SIGUSR1, sig_usr1_handler)** al padre prima della creazione dei figli -> il figlio eredita l'associazione segnale-handler. (risolve con certezza il problema, ma va bene solo se il padre non ha bisogno di gestire diversamente SIGUSR1)
 - Oppure introdurre una sincronizzazione figli-padre prima dell'invio di **SIGUSR1** :
-

```

int OKF=0;
int main(int argc, char* argv[]) {
    int i, n, pid[MAX_CHILDREN];
    n = atoi(argv[1]);
    k=atoi(argv[2]);
    signal(SIGUSR2, figlio_ok);
    for(i=0; i<n; i++) {
        pid[i] = fork();
        ...
    }
    while(OKF<k) pause(); //figli pronti
    for (i=0; i<k; i++) kill(pid[i], SIGUSR1);
    for (i=0; i<n; i++) wait_child();
    return 0;
}
..
void wait_for_signal(){
    signal(SIGUSR1, sig_usr1_handler);
    kill(getppid(), SIGUSR2); //figlio pronto
    ...}

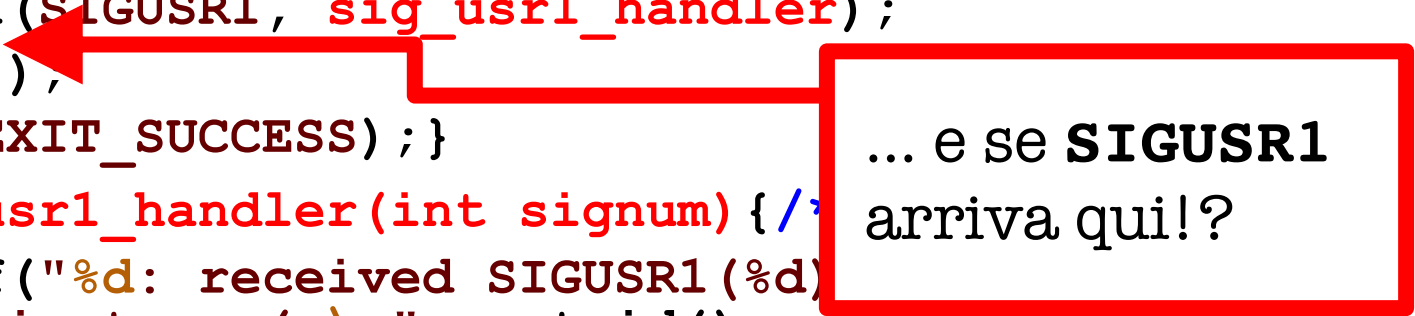
```

```
void figlio_ok(int signum) {  
    OKF++;  
    printf("figlio %d -simo pronto\n", OKF);  
}
```

NB: Questa soluzione risolve con certezza il problema solo in caso di modello affidabile dei segnali, in cui (contrariamente a quanto accade in linux) tutti i segnali ricevuti da un processo sono opportunamente accodati e non vengono mai accorpati

Esempio - Riflessione B

```
void wait_for_signal() {  
    /* Imposto il gestore dei segnali di tipo SIGUSR1 */  
    signal(SIGUSR1, sig_usr1_handler);  
    pause();  
    exit(EXIT_SUCCESS);}  
  
void sig_usr1_handler(int signum) {  
    printf("%d: received SIGUSR1(%d)  
        terminate :-( \n", getpid(), signum);  
}  
  
void sleep_and_terminate() {  
    sleep(5);  
    printf("%d: Slept 5sec. Withdrowing.\n",getpid());  
    exit(EXIT_SUCCESS);}  
  
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 - 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 nell' handler **TUTTE** le operazioni che il figlio deve fare alla ricezione del segnale, **inclusa la exit** :

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

Esercizio 1

Si scriva un programma C con la seguente interfaccia:

./saluta N

il processo P0 deve creare un figlio P1 il cui compito è di stampare ripetutamente (a intervalli di un secondo) la stringa «Hello world» seguita dal valore di un contatore X (inizializzato a 0) che indica il numero di ripetizione.

Pertanto alla prima ripetizione, P1 dovrà stampare «Hello world 0», alla seconda «Hello world 1», e così via.

Dopo **N** secondi, P1 deve interrompere ciò che sta facendo, inviare un segnale P0 e infine terminare.

Nel frattempo P0 attende senza far niente. Al termine di P1, P0 stamperà le informazioni relative alla terminazione del figlio.

Esercizio 1 – Nota

Dopo **N** secondi P1 deve smettere di fare quel che sta facendo e inviare un segnale a P0.

P1 ha un lavoro da compiere. Non può attendere **N** secondi senza far nulla...

=> occorre una primitiva che imposti un timeout senza sospensione...!

Esercizio 2

Si scriva un programma C con la seguente interfaccia:

./repeat_wave N

il processo P0 deve creare un figlio P1 il cui compito è lanciare ripetutamente all'infinito (a intervalli di 1 secondo) un programma «Hello world».

«Hello world» deve avere la seguente interfaccia:

./hello x (dove **x** è un intero positivo)

Alla prima invocazione, P1 dovrà lanciare **./hello 0** per poi incrementare il valore di **x** ad ogni successiva invocazione.

Dopo **N** secondi, P1 deve interrompere ciò che sta facendo, inviare un segnale P0 e infine terminare.

Nel frattempo P0 attende senza far niente.

Esercizio 2 - Nota

P1 deve lanciare **hello** a intervalli di 1 secondo.
=> ho bisogno di un ciclo opportuno di **exec()**

Ma...

La **exec** sostituisce codice e dati del processo chiamante:

```
execl("/home/daniela/pippo", "pippo", "arg1" (char*)0);  
perror("Errore in execl\n");  
exit(1);
```

Può P1 eseguire **hello** , dormire 1 secondo e poi rieseguirlo?

Può far fare **hello** a qualcun'altro?

Devo generare ALMENO P0 e P1, ma non sono obbligato a generare solo loro!

Esercizio 3

Si scriva un programma C con la seguente interfaccia:

./launcher COMMAND PARAM

dove:

- **COMMAND** è un comando bash che prende in ingresso un solo parametro
- **PARAM** è il parametro di **COMMAND**

Quindi ad esempio **COMMAND PARAM** potrebbero assumere valore “**ls Desktop**” oppure “**cat myfile**”, ecc...

Il processo P0 genera due figli P1 (controllore) e P2 (esecutore).

Esercizio 3

- **P2 (esecutore)** deve eseguire il **COMMAND PARAM** e inviare a P1 un diverso segnale a seconda dell'esito di tale esecuzione:
 - ☐ Se è fallita (es: perchè **PARAM** è scorretto o perche **COMMAND** non esiste), deve inviare **SIGUSR1**
 - ☐ Se l'esecuzione è andata bene, deve inviare **SIGUSR2**
- **P1 (controllore)** attende senza far nulla il segnale da P2 e stampa a video “*ok – esecuzione avvenuta con successo*” oppure “*no – ho rilevato un problema*” a seconda del segnale ricevuto

Esercizio 3 – Nota 1

- L'esecuzione di **COMMAND PARAM** può fallire per due motivi:
 - ❑ **COMMAND** non esiste => la exec non va a buon fine
 - ❑ **PARAM** non è corretto (es: “**ls Deskt**”) => la exec va a buon fine (perchè **ls** esiste), ma **COMMAND** termina con uno stato di terminazione diverso da 0 (perchè **Deskt** non esiste).
 - In entrambi i casi P2 deve inviare a P1 il segnale **SIGUSR1**
-

Esercizio 3 – Riflessioni

- P2 deve lanciare **COMMAND** e poi inviare un segnale, ma la **exec** non ha ritorno se il lancio va a buon fine
⇒ si pone lo stesso problema dell'esercizio 1
 - Potremmo invertire il ruolo dei figli? Potremmo cioè nominare P1 esecutore e P2 controllore?
=> si noti che in questo caso P1 dovrebbe inviare un segnale a P2... Per farlo dovrebbe conoscerne il PID!
-