

Esercitazione 3 - Gruppo LZ

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• <code>kill -1</code> 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 (in secondi) specificato come argomento. Ritorna il numero di secondi mancante allo scadere del time-out precedente. Chiamata non bloccante.
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 (loop)
 - 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. Withdrawing.\n",getpid());
    exit(EXIT_SUCCESS);}
```

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

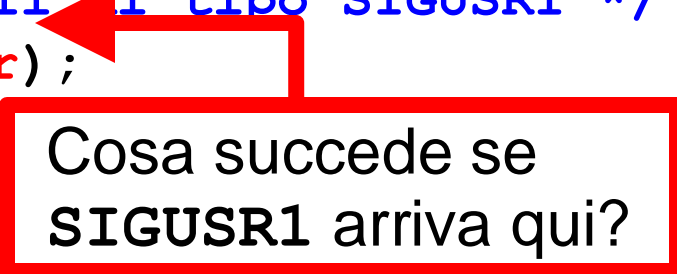
Esempio – Riflessione A

```
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). Will
        terminate :-( \n", getpid(), signum);}

void sleep_and_terminate() {
    sleep(5);
    printf("%d: Slept 5sec. Withdrawing.\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**, così che P0 invii **SIGUSR1** solo quando è certo che tutti i figli abbiano impostato un handler per tale segnale
-

```

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();
}

void wait_for_signal(){
    signal(SIGUSR1, sig_usr1_handler);
    kill(getppid(), SIGUSR2); //figlio pronto
    ...}

```

```

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

```

Esempio – Riflessione A

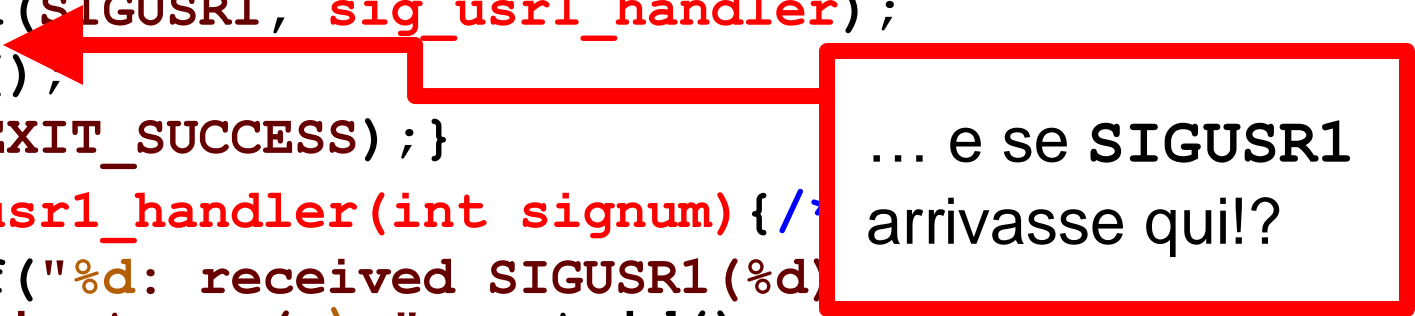
Sincronizzazione padre-figlio:

- P0 invia **SIGUSR1** solo quando ha ricevuto **OKF=k** segnali **SIGUSR2**
- Ciascun figlio invia un **SIGUSR2** al padre dopo aver impostato il suo handler per **SIGUSR1**
- Pertanto, si ha la garanzia che l'handler corretto (**sig_usr1_handler**) sia stato impostato per tutti i figli nel momento in cui P0 inizia a inviare i **SIGUSR1**

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. Withdrawing.\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
arrivasse 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 (1/2)

Si scriva un programma C con la seguente interfaccia:

./sleepAndLaunch S R

dove:

- **S** è un intero positivo
- **R** è un intero in $[0,1]$

Il processo P0 deve creare un figlio P1 e attendere la sua terminazione.

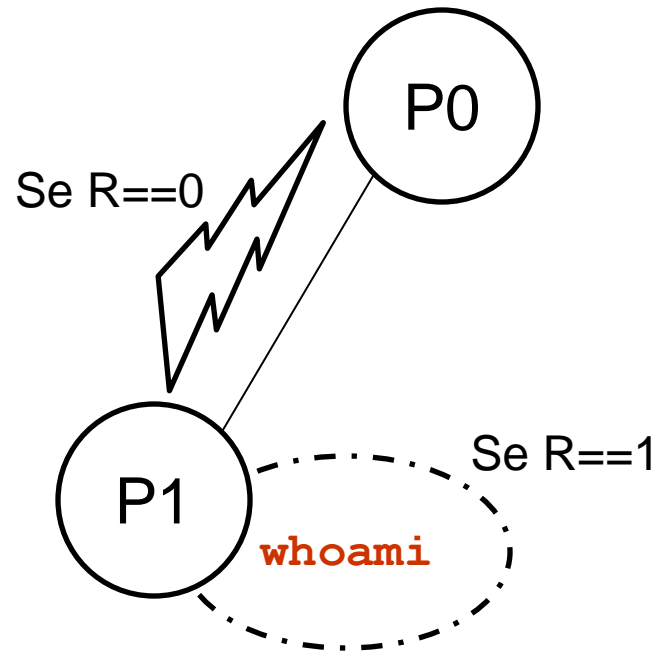
Il processo P1 deve inizialmente attendere **S** secondi e poi controllare il parametro **R** passato come argomento:

- Se **R==0**, P1 deve inviare un segnale a P0 e poi terminare
 - Se **R==1**, P1 deve lanciare il comando **whoami**, stampare a video la stringa «Comando whoami eseguito con successo» e poi terminare
-

Esercizio 1 (2/2)

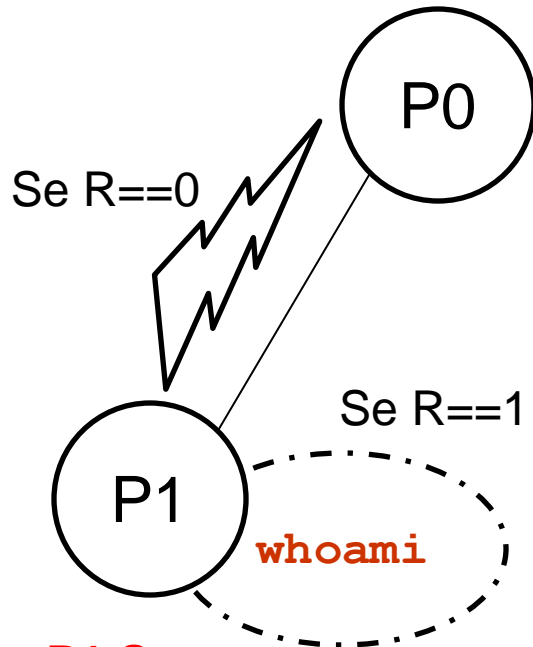
- Alla (eventuale) ricezione del segnale da P1, P0 deve stampare la stringa <<Ricevuto il segnale da P1!>> e terminare.
- P0 deve inoltre controllare che l'esecuzione non superi mai i **3 secondi**. Pertanto, trascorso tale tempo dall'inizio della sua esecuzione, P0 deve stampare la stringa <<Timeout scaduto!>>, terminare P1 e terminare a sua volta.

Esercizio 1 – Riflessioni (1/2)



- **P1 deve attendere S secondi senza far nulla:** quale primitiva?
 - **P0 controllare che l'esecuzione non superi mai i 3 secondi.**
Pertanto, deve attendere **3** secondi prima di stampare <<Timeout scaduto!>>, ma ne frattempo deve fare altre cose (creare P1, attendere che termini P1, ecc...)Quale primitiva?
-

Esercizio 1 – Riflessioni (2/2)



P1 Stampa:
Comando eseguito
correttamente!

- Se l'esecuzione del comando **whoami** avviene con successo, P1 deve stampare a video un messaggio e poi terminare.

Per lanciare **whoami** ho bisogno di una `exec()`, **ma...**

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

```
execlp(comando, comando, char*)0);  
perror("Errore in execl\n");  
exit(1);
```

Può P1 eseguire comando, e poi fare una `printf`?

Può far eseguire comando a qualcun altro?

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

Esercizio 2

Realizzare una variante dell'esercizio 1 in cui:

- Il processo P0 crea P1 e, intanto che attende la sua terminazione, stampa continuamente a intervalli di un secondo:

$$P0: e^0 = 1$$

$$P0: e^1 = 2,71$$

$$P0: e^2 = 7,38$$

$$P0: e^3 = 20,08$$

...

Esercizio 2 – Riflessioni

- Calcolo delle potenze del numero di Nepero:

```
#include <math.h>
```

```
exp(x) ; //calcola  $e^x$ 
```

- **P0** deve stampare continuamente: non può sospendersi in attesa della terminazione dei figli senza far nulla (wait)

→ Gestione del segnale **SIGCHLD**.

- ogni figlio che termina provoca l'invio del segnale SIGCHLD al padre;
- il trattamento di default per SIGCHLD è SIG_IGN;
- per gestire il segnale in modo diverso, è necessario agganciare un handler al segnale:

```
signal(SIGCHLD, myhandler) ;
```