

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• <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);}

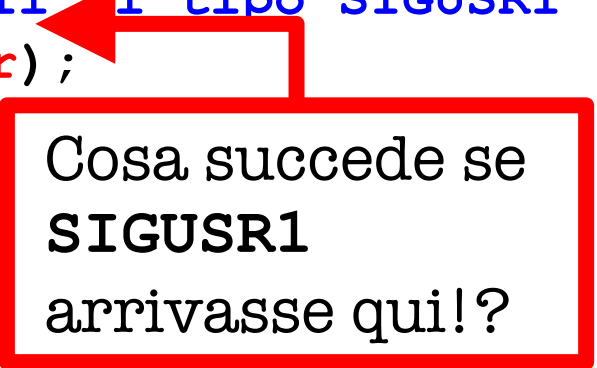
void sig_usr1_handler(int signum) { /*Gestione segnale*/
    printf("%d: received SIGUSR1(%d). Will
        terminate :-( \n", getpid(), signum);}
```

```
void sleep_and_terminate() {
    sleep(5);
    printf("%d: Slept 5sec. 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)  
        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
arrivasse 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
    ...}

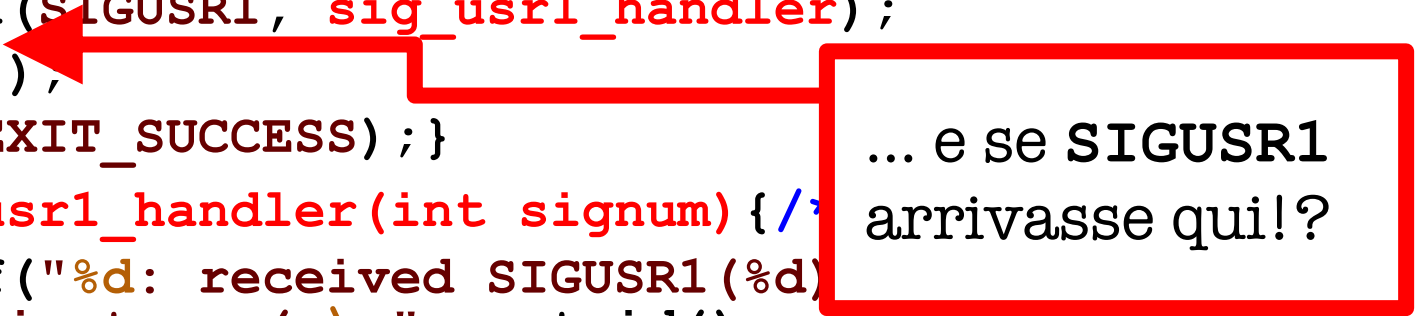
```

```
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
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 (1/3)

Si scriva un programma C con la seguente interfaccia:

./es3 COM T

dove:

- COM è una stringa (corrispondente a un comando unix)
- T è un intero positivo

Il processo P0 deve creare due figli P1 e P2.

Successivamente, P0 stampa all'infinito ad intervalli di 1 secondo il risultato di 2^n per n crescente in $[0, \infty[$.

Esempio:

$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

...

P1 invece stampa all'infinito a intervalli di 1 secondo il risultato di \sqrt{n} (funzione `sqrt()` in `math.h`) per n crescente in $[1, \infty[$.

Esempio:

`sqrt(1) = 1`

`sqrt(2) = 1,4142135`

...

Esercizio 1 2/3)

- Il processo P2 deve inizialmente dormire 3 secondi e poi controllare il proprio PID.
 - ☐ Se il PID è pari, deve inviare un segnale a P0 e terminare
 - ☐ Se il PID è dispari, deve inviare un segnale a P1 e terminare
- Alla (eventuale) ricezione del segnale, P0 deve stampare la stringa <<Finito!>>, terminare entrambi i figli e terminare a sua volta.
- Alla (eventuale) ricezione del segnale, P1 deve invece lanciare il comando **COM** passato a linea di comando.

Esercizio 1 (3/3)

- P0 deve inoltre:
 - ☐ gestire la terminazione dei figli stampando a video la stringa <<Figli terminati!>> una volta che entrambi hanno concluso l'esecuzione.
 - ☐ In ogni caso, trascorsi **T** secondi dall'inizio dell'esecuzione P0 deve stampare la stringa <<Timeout scaduto!>>, terminare entrambi i figli e terminare a sua volta.

Esercizio 1 - Note

P0 Stampa:

$$2^0 = 1$$

$$2^1 = 2$$

$$2^2 = 4$$

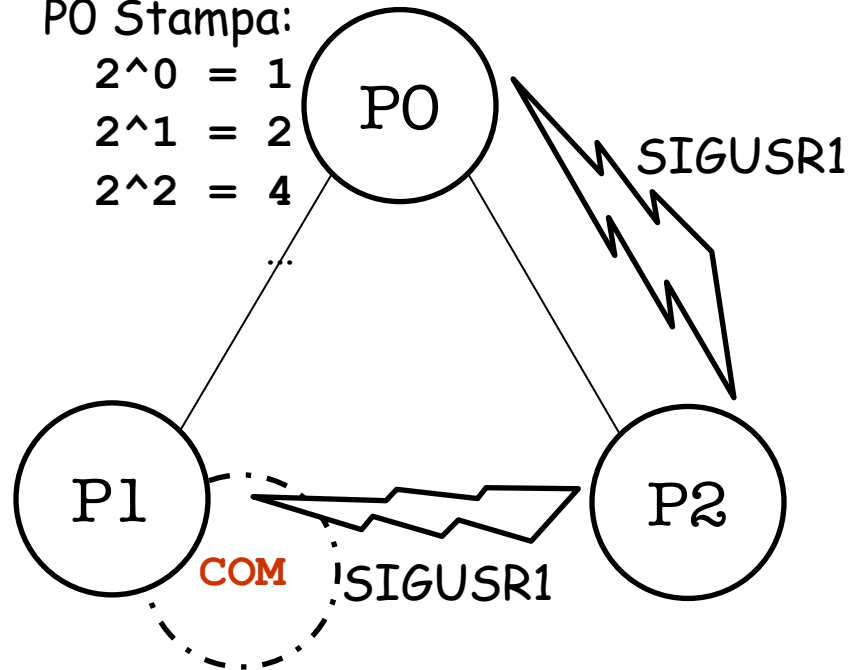
...

P1 Stampa:

$$\text{sqrt}(1) = 1$$

$$\text{sqrt}(2) = 1,41421$$

...



- **P2 manda un segnale a P1.** Potremmo invertire il ruolo dei figli? Cioè: P1 potrebbe mandare un segnale a P2 ?
- **P0 e P1** devono stampare una volta al secondo: come temporizzare le printf? → alarm o sleep?

Esercizio 1 - Note

P0 Stampa:

$$2^0 = 1$$

$$2^1 = 2$$

$$2^2 = 4$$

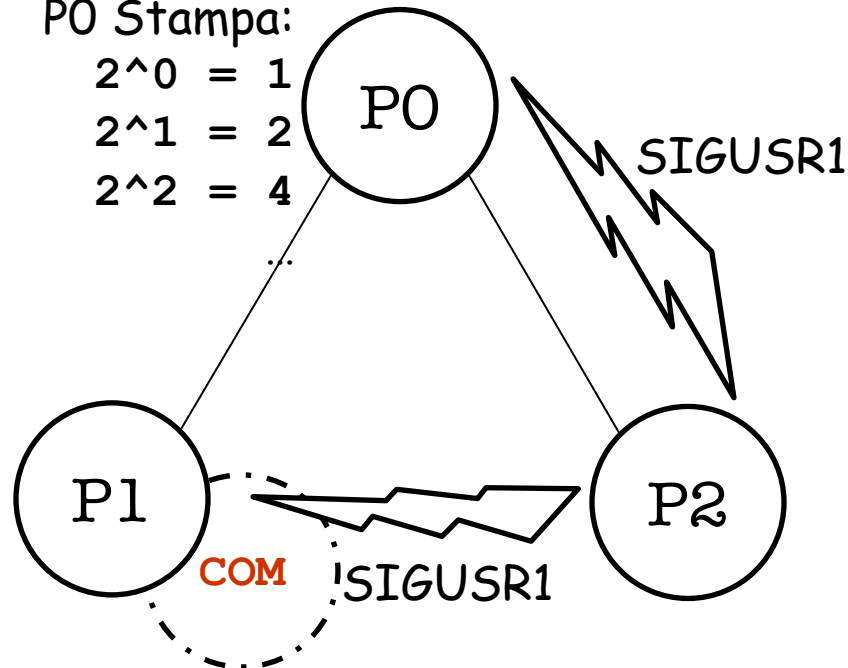
...

P1 Stampa:

$$\text{sqrt}(1) = 1$$

$$\text{sqrt}(2) = 1,41421$$

...



- **P0** deve stampare continuamente: non può sospendersi in attesa dei figli senza far nulla... => gestione del segnale **SIGCHLD**.
- **P0** deve stampare continuamente e contemporaneamente attendere **T** secondi prima di stampare <<Timeout scaduto!>>

Gestione SIGCHLD

Ricordare:

- 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, handler);
```

Esercizio 2

Realizzare una variante dell'esercizio 1 in cui:

- Alla ricezione del segnale, P1 deve lanciare **COM** (come nell'esercizio 1)
 - Alla fine dell'esecuzione di **COM**, P1 stampa la stringa <<Comando <**COM**> eseguito correttamente!>>
-

Esercizio 2 - Nota

Per lanciare **COM** ho bisogno di una **exec()**

Ma...

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

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

Può P1 eseguire **COM**, e poi fare una printf?

Può far eseguire **COM** a qualcun altro?

Devo generare ALMENO P0, P1 e P2, ma non sono obbligato a generare solo loro!

Esercizio 2 - Nota

P0 Stampa:

$$2^0 = 1$$

$$2^1 = 2$$

$$2^2 = 4$$

...

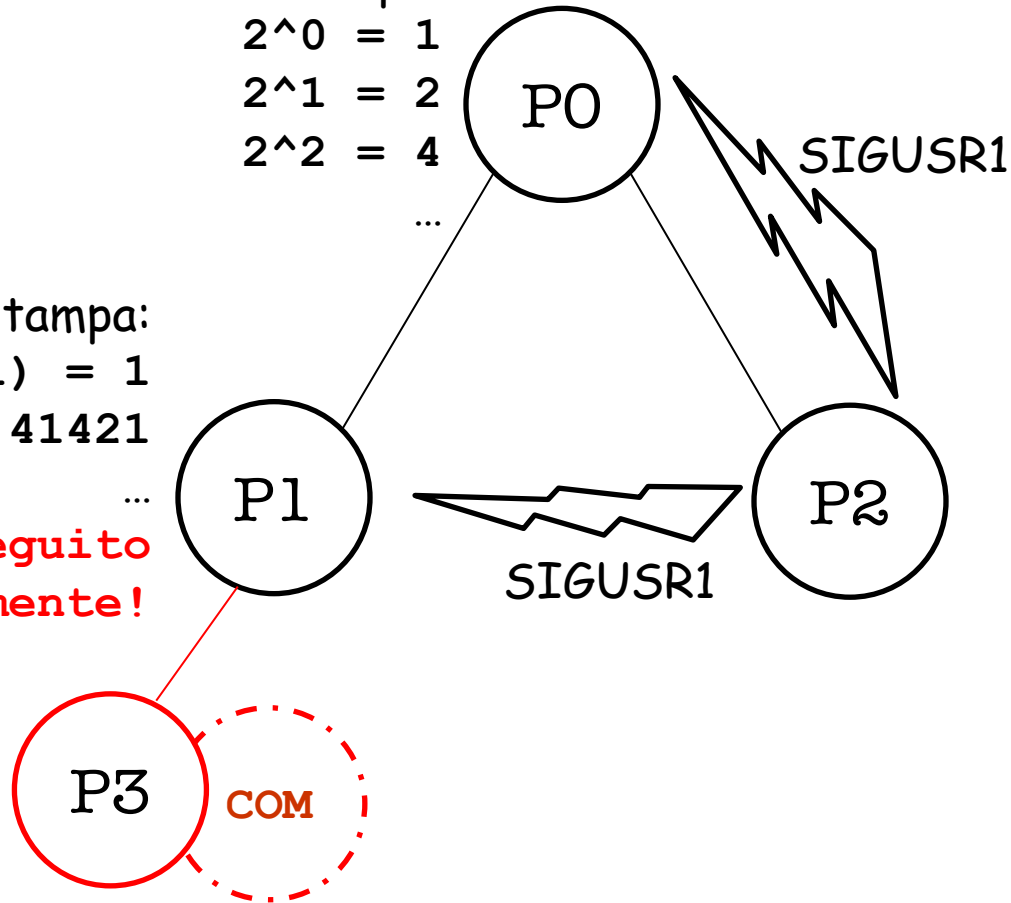
P1 Stampa:

$$\text{sqrt}(1) = 1$$

$$\text{sqrt}(2) = 1,41421$$

...

Comando <COM> eseguito
correttamente!



P1 creerà un nipote, dedicato all'esecuzione del comando COM.
Terminata l'esecuzione del nipote, P1 farà la printf.
