Terza Esercitazione

Gestione di segnali in Unix

Primitive `signal` e `kill`

Primitive fondamentali (sintesi)

`signal`	• **Imposta la reazione del processo** all'eventuale ricezione di un segnale (può essere una funzione handler, SIG_IGN o SIG_DFL)
`kill`	• **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`	• Chiamata **bloccante**: il processo si sospende fino alla ricezione di un qualsiasi segnale
`alarm`	• "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`	• **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). 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) */
    …}
```

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
          pronto\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. 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:

```
./segnali com
```

dove:

- `com` è una stringa che si suppone corrisponda ad un comando di shell

Il processo P0 deve creare due figli, P1 e P2 e poi generare randomicamente (si veda la funzione rand) un numero intero **X** compreso tra 0 e 4 (estremi inclusi).

- Il processo P1 deve inizialmente dormire 3 secondi e poi:
  - Se **X** è dispari deve inviare un segnale a **P0**
  - Se **X** è pari deve inviare un segnale a **P2**
- Infine il processo P1 deve terminare

Esercizio 1 (2/3)

P0, deve inoltre attendere la terminazione dei figli e, contemporaneamente, stampare all'infinito ad intervalli di 1 secondo il messaggio:

P0 (PID=<pid_di_p0>): attendo il segnale da <count> secondi

Esempio:

P0 (PID=1234): attendo un segnale da 0 secondi
P0 (PID=1234): attendo un segnale da 1 secondi
P0 (PID=1234): attendo un segnale da 2 secondi
….

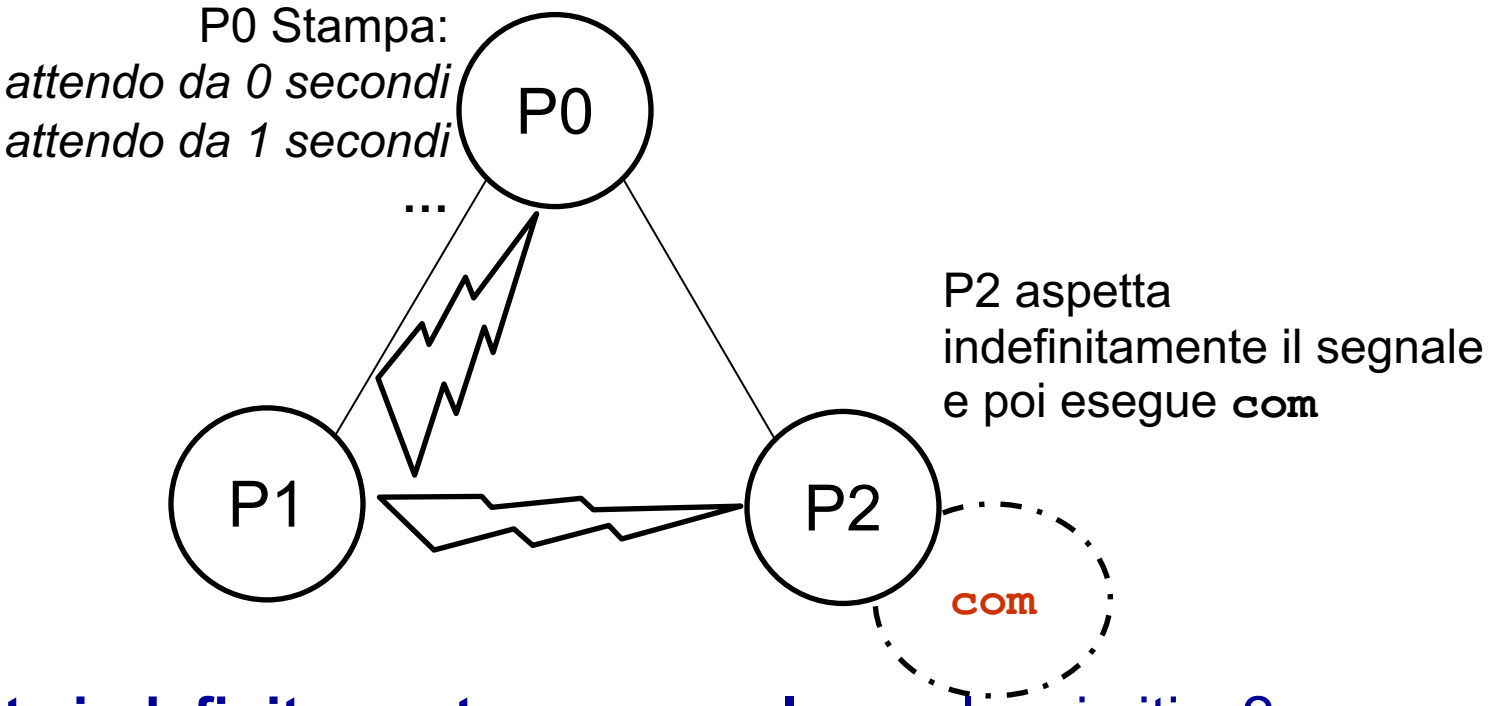
- Alla ricezione del segnale da P1, P0 deve stampare la stringa <<Ricevuto segnale da P1!>>, terminare entrambi i figli e terminare a sua volta.
- In ogni caso, trascorsi **X** secondi dall'inizio dell'esecuzione P0 deve stampare la stringa <<Timeout di **X** secondi scaduto!>>, terminare entrambi i figli e terminare a sua volta.

Esercizio 1 (3/3)

- P2 deve attendere (indefinitamente, senza far altro) il segnale proveniente da P1. Alla ricezione del segnale, P2 deve lanciare il comando `com` passato da linea di comando e terminare

- P0 deve infine gestire la terminazione dei figli stampando a video la stringa <<Figli terminati!>> una volta che entrambi hanno concluso l'esecuzione

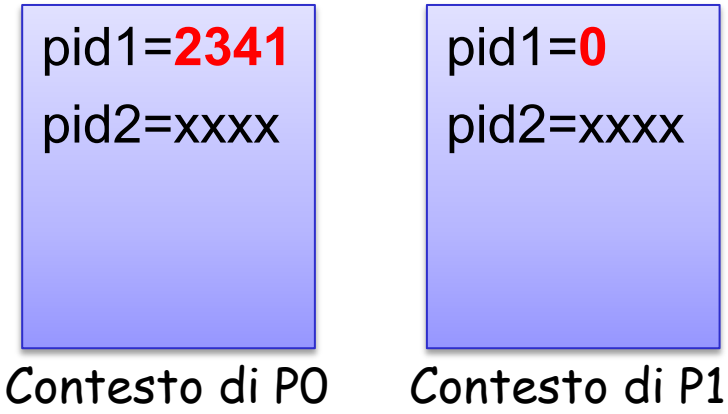
Esercizio 1 – Riflessioni (1/3)



- **P2 aspetta indefinitamente un segnale**: quale primitiva?
- **P0** deve stampare una volta al secondo: come temporizzare le printf? → alarm o sleep?
- **P1 manda un segnale a P2**. **Si può fare davvero??**

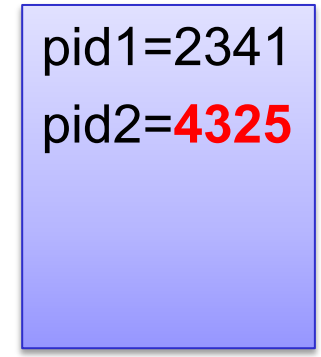
P1 può mandare un segnale a P2?

```
int pid1,pid2;
pid1 = fork();
if (pid1==0){
   //codice di P1
} else if (pid1>0){
   //codice di P0
   pid2 = fork();
   if (pid2==0){
      //codice di P2
   } else if (pid2>0){
      //codice di P0
   } else {//errore nella fork()}
} else {//errore nella fork()}
```

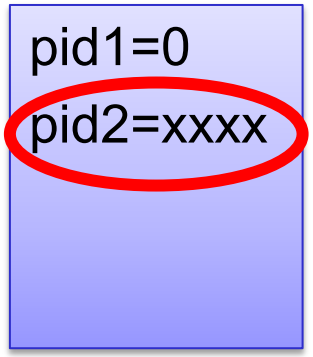


P1 può mandare un segnale a P2?

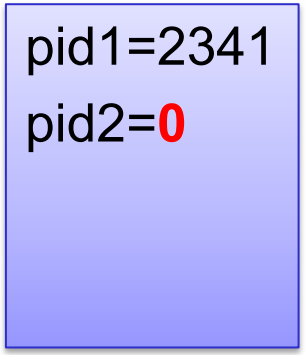
```
int pid1,pid2;
pid1 = fork();
if (pid1==0){
   //codice di P1
} else if (pid1>0){
   //codice di P0
   pid2 = fork();
   if (pid2==0){
      //codice di P2
   } else if (pid2>0){
      //codice di P0
   } else {//errore nella fork()}
} else {//errore nella fork()}
```



Contesto di P0

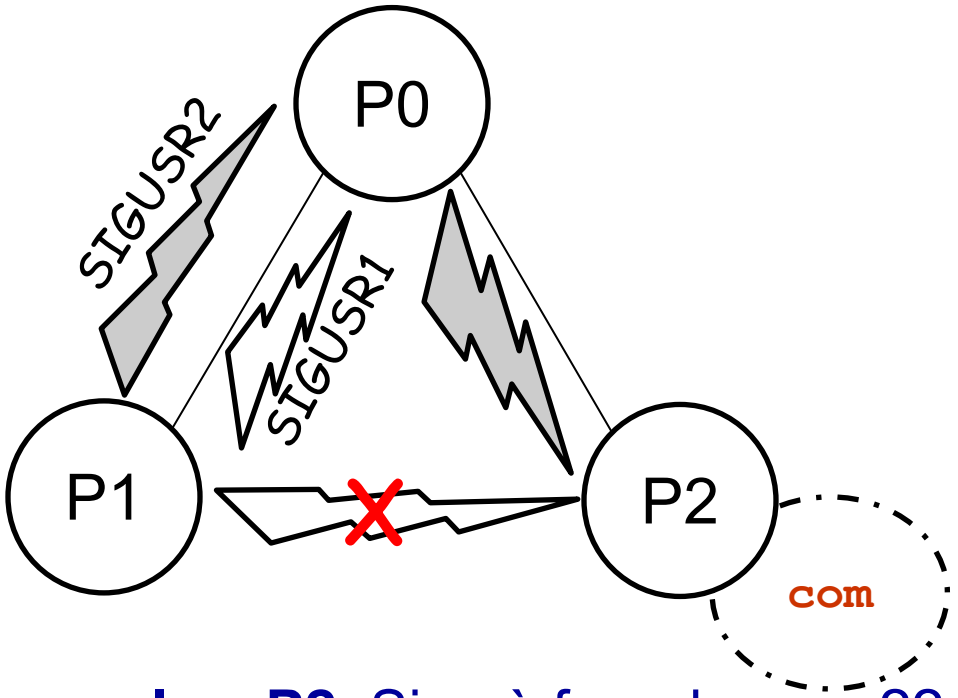


Contesto di P1



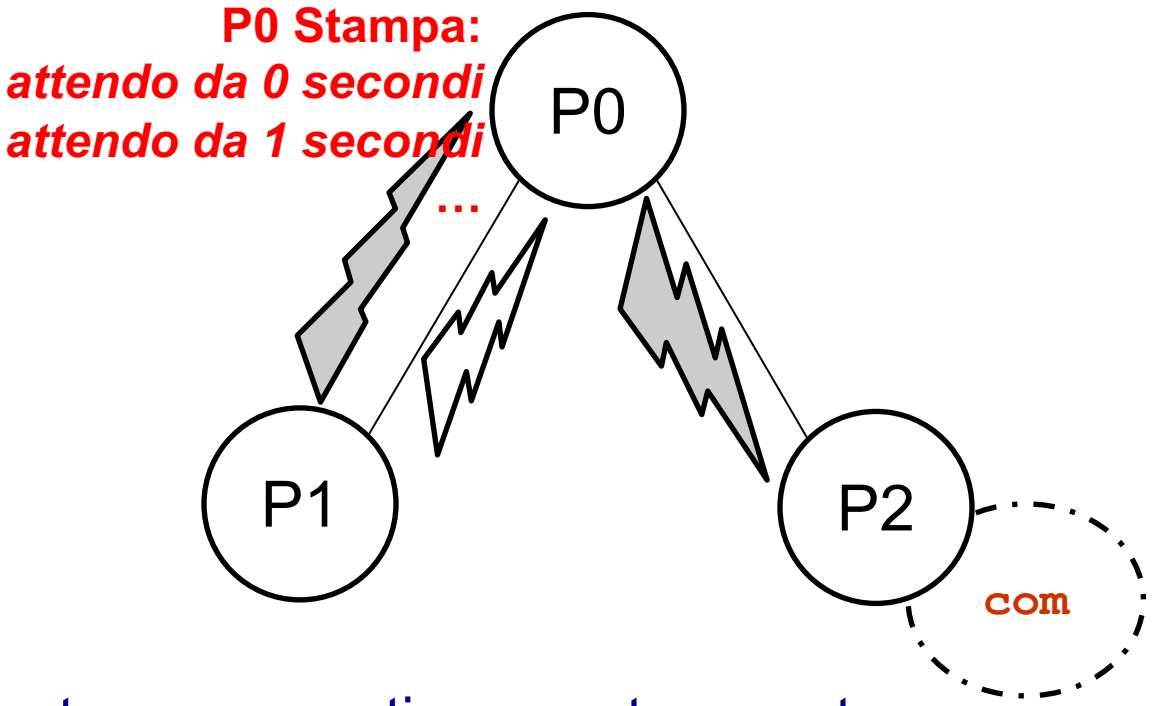
Contesto di P2

Esercizio 1 – Riflessioni (2/3)



- **P1 manda un segnale a P2**. Si può fare davvero?? => **NO**, perché al momento della creazione di P1, il pid di P2 non è ancora stato inizializzato
- Se P1 deve notificare P2, dobbiamo passare dal padre P0
- Poiché P0 ora riceve due segnali occorre poterli distinguere

Esercizio 1 – Riflessioni (3/3)



- **P0** deve stampare continuamente e contemporaneamente attendere **X** secondi prima di stampare <<Timeout di **X** secondi scaduto!>> Quale primitiva?
- **P0** deve stampare continuamente: non può sospendersi in attesa della terminazione dei figli senza far nulla (wait) ➔ gestione del segnale **SIGCHLD.**

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(SIGHCHLD, handler);
```

Esercizio 2

Realizzare una variante dell'esercizio 1 in cui:

- Alla ricezione del segnale, P2 deve lanciare `com` passato da linea di comando (come nell'esercizio 1)
- Alla fine dell'esecuzione di `com`, SOLO se tale comando è stato eseguito con successo, P2 stampa il messaggio:

  Comando <com> eseguito correttamente!

Esercizio 2 – Riflessioni (1/2)

Per lanciare **`date`** ho bisogno di una **`exec()`**

Ma…

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

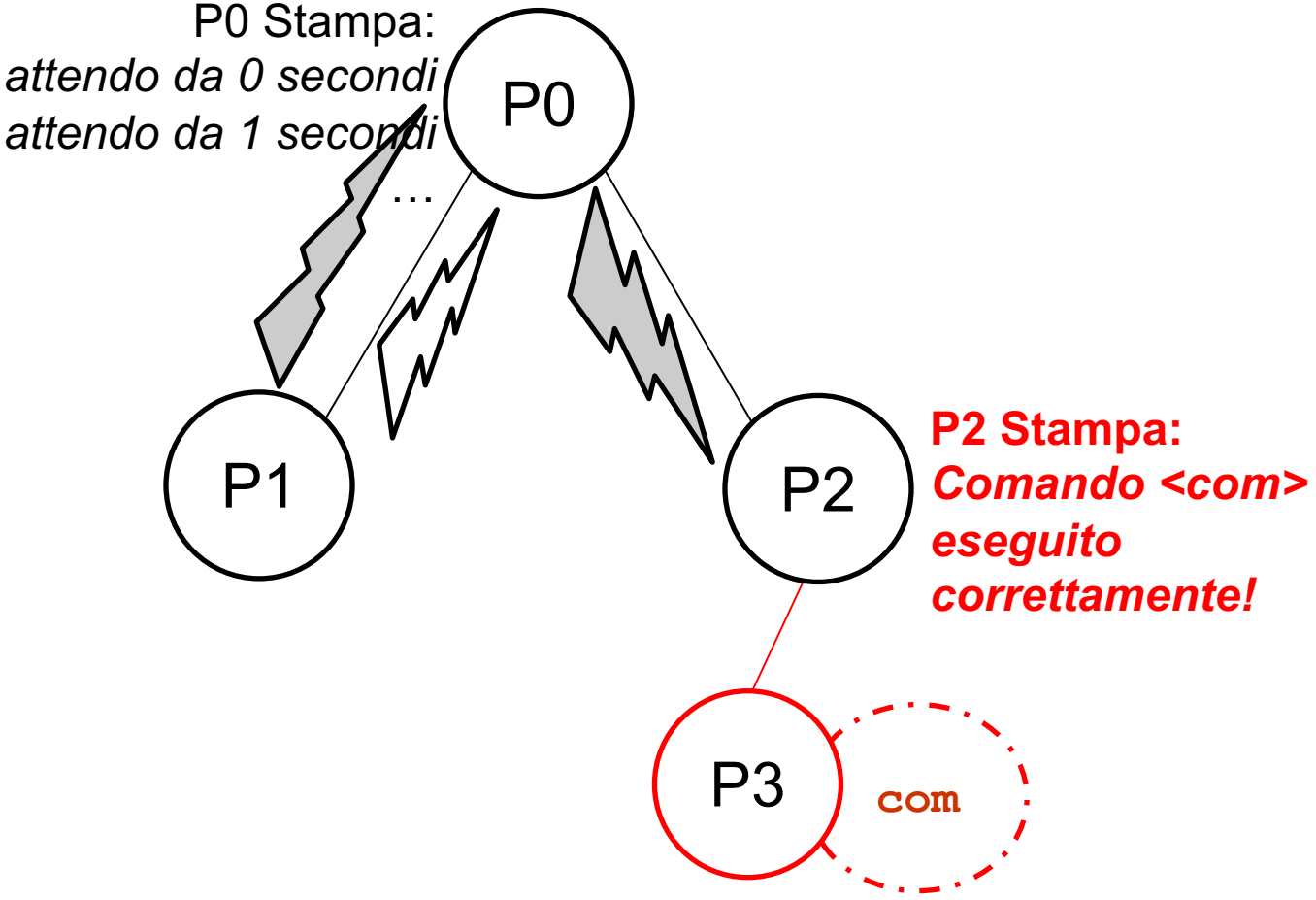
```
execlp(com,com, char*)0);
perror("Errore in execl\n");
exit(1);
```

Può P2 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 – Riflessioni (2/2)



- P2 creerà un nipote P3, dedicato all'esecuzione del comando `com`. Terminata l'esecuzione del nipote, P2 farà la printf.