Seconda Esercitazione

Gestione dei processi in Unix
Primitive fork, wait, exec

Andrea Reale
andrea.reale@unibo.it

Prima di iniziare...

- Come sapete, accesso **temporaneo** a lab4 con:
  - username: lab4-XX (XX è il numero della macchina)
  - password: lab4-XX
- Per ottenere il **vostro** account definitivo, utilizzabile anche negli altri laboratori:
  1. Accedere alla macchina con il login temporaneo
  2. Collegarsi a **http://infoy.ing.unibo.it/**
  3. Creare un account
- In caso di problemi, potrete fare la stessa procedura dal **Lab0** dopo l'esercitazione

Tabella Riassuntiva

fork	• Generazione di un processo figlio, che **condivide il codice** con il padre e possiede **dati replicati** • **Restituisce** il PID del processo creato per il padre, 0 per il figlio, o un valore negativo in caso di errore
exit	• **Terminazione** di un processo • Accetta come parametro lo **stato di terminazione.** Per convenzione 0 indica un'uscita con **successo**, un valore *non-zero* indica uscita con **fallimento**.
wait	• Chiamata **bloccante:** raccolta dello stato di terminazione di un figlio • Restituisce il PID del figlio terminato e permette di capire il **motivo della terminazione** (es. volontaria? con quale stato? Involontaria? A causa di quale segnale?)
exec	• **Sostituzione di codice e dati** di un processo • **NON** crea processi figli

Un esempio - fork() e exit()

- Scrivere un programma in cui il processo padre proceda all'instanziazione di un numero N di figli, **tutti fratelli**.
- Il secondo parametro può assumere il valore '1' o '0'
  - Se '1' i figli terminano "correttamente" facendo la exit()
  - Se '0' i figli terminano senza fare la exit()
- Interfaccia:

$ ./generate <N> <term>

- **Obiettivo**
  - Cosa succede se un figlio di "dimentica" di fare la exit()?

Esempio - Il Codice

```
void main(int argc, char* argv[]) {
   int i, j, k, pid, status, n_children;
   char term;
   n_children = atoi(argv[1]);
   term = argv[2][0];
   for ( i=0; i<n_children; i++ ) {
      pid = fork();
      if ( pid == 0 ) { // Eseguito dai figli
         if ( term == '1' )
            exit(0);
      }
      else if ( pid > 0 ) { // Eseguito dal padre
         printf("%d: child created with PID %d\n", getpid(), pid);
      }
      else {
         perror("Fork error:");
         exit(1);
      }
   }
}
```

Simulazione di Esecuzione (1/7)

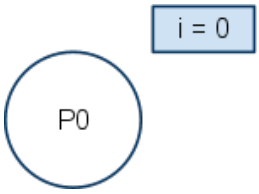
- Vediamo cosa succede in un'esecuzione del programma
- Assumiamo:
  - **N = 2** : Il padre genera due processi figli
  - **term = '0'** : I figli non terminano una volta creati
- Da ricordare:
  - Una volta creati i figli eseguono **concorrentemente** al padre e tra di loro.

Processo P0

```
void main(int argc, char* argv[]) {
   int i, j, k, pid, status, n_children;
   char term;
   n_children = atoi(argv[1]);
   term = argv[2][0];
   for ( i=0; i<n_children; i++ ) {
      pid = fork();
      if ( pid == 0 ) { // Eseguito dai figli
         if ( term == '1' )
            exit(0);
      }
      else if ( pid > 0 ) { // Eseguito dal padre
         printf("%d: child created with PID %d\n", getpid(), pid);
      }
      else {
         perror("Fork error:");
         exit(1);
      }
   }
}
```

i = 0

Simulazione di Esecuzione (2/7)

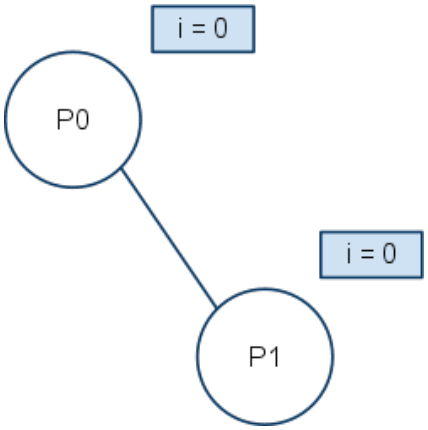


Processo P0

```
void main(int argc, char* argv[]) {
   int i, j, k, pid, status, n_children;
   char term;
   n_children = atoi(argv[1]);
   term = argv[2][0];
   for ( i=0; i<n_children; i++ ) {
      pid = fork();
      if ( pid == 0 ) { // Eseguito dai figli
         if ( term == '1' )
            exit(0);
      }
      else if ( pid > 0 ) { // Eseguito dal padre
         printf("%d: child created with PID %d\n", getpid(), pid);
      }
      else {
         perror("Fork error:");
         exit(1);
      }
   }
}
```

i = 0

Simulazione di Esecuzione (3/7)



Simulazione di esecuzione (4/7)

*Continuiamo a concentrarci su **P0** (padre)*
*Per il momento trascuriamo **P1**, che intanto sta eseguendo...*

Processo **P0**

```
void main(int argc, char* argv[]) {
   int i, j, k, pid, status, n_children;
   char term;
   n_children = atoi(argv[1]);
   term = argv[2][0];
   for ( i=0; i<n_children; i++ ) {
      pid = fork();
      if ( pid == 0 ) { // Eseguito dai figli
         if ( term == '1' )
            exit(0);
      }
      else if ( pid > 0 ) { // Eseguito dal padre
         printf("%d: child created with PID %d\n", getpid(), pid);
      }
      else {
         perror("Fork error:");
         exit(1);
      }
   }
}
```

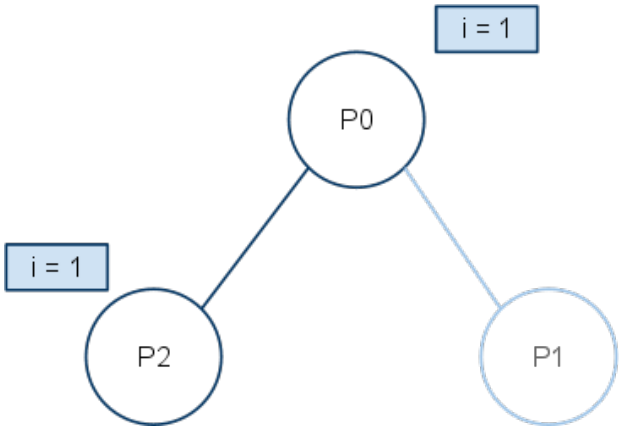
i = 0

Processo **P0**

```
void main(int argc, char* argv[]) {
   int i, j, k, pid, status, n_children;
   char term;
   n_children = atoi(argv[1]);
   term = argv[2][0];
   for ( i=0; i<n_children; i++ ) {
      pid = fork();
      if ( pid == 0 ) { // Eseguito dai figli
         if ( term == '1' )
            exit(0);
      }
      else if ( pid > 0 ) { // Eseguito dal padre
         printf("%d: child created with PID %d\n", getpid(), pid);
      }
      else {
         perror("Fork error:");
         exit(1);
      }
   }
}
```

i = 1

SImulazione di Esecuzione (5/7)



Processo P0

```
void main(int argc, char* argv[]) {
   int i, j, k, pid, status, n_children;
   char term;
   n_children = atoi(argv[1]);
   term = argv[2][0];
   for ( i=0; i<n_children; i++ ) {
      pid = fork();
      if ( pid == 0 ) { // Eseguito dai figli
         if ( term == '1' )
            exit(0);
      }
      else if ( pid > 0 ) { // Eseguito dal padre
         printf("%d: child created with PID %d\n", getpid(), pid);
      }
      else {
         perror("Fork error:");
         exit(1);
      }
   }
}
```

i = 1

Processo P0

```
void main(int argc, char* argv[]) {
   int i, j, k, pid, status, n_children;
   char term;
   n_children = atoi(argv[1]);
   term = argv[2][0];
   for ( i=0; i<n_children; i++ ) {
      pid = fork();
      if ( pid == 0 ) { // Eseguito dai figli
         if ( term == '1' )
            exit(0);
      }
      else if ( pid > 0 ) { // Eseguito dal padre
         printf("%d: child created with PID %d\n", getpid(), pid);
      }
      else {
         perror("Fork error:");
         exit(1);
      }
   }
}
```

i = 2

Simulazione di esecuzione (6/7)

- **P0** a questo punto ha creato tutti i figli che doveva

MA

- Cosa hanno fatto i suoi figli nel frattanto?
- Iniziamo da **P2**...
  - Ricordate: i processi figli non terminano subito dopo essere stati creati (term = '0')

Processo P2

```
void main(int argc, char* argv[]) {
   int i, j, k, pid, status, n_children;
   char term;
   n_children = atoi(argv[1]);
   term = argv[2][0];
   for ( i=0; i<n_children; i++ ) {
      pid = fork();
      if ( pid == 0 ) { // Eseguito dai figli
         if ( term == '1' )
            exit(0);
      }
      else if ( pid > 0 ) { // Eseguito dal padre
         printf("%d: child created with PID %d\n", getpid(), pid);
      }
      else {
         perror("Fork error:");
         exit(1);
      }
   }
}
```

i = 1

Processo P2

```
void main(int argc, char* argv[]) {
   int i, j, k, pid, status, n_children;
   char term;
   n_children = atoi(argv[1]);
   term = argv[2][0];
   for ( i=0; i<n_children; i++ ) {
      pid = fork();
      if ( pid == 0 ) { // Eseguito dai figli
         if ( term == '1' )
            exit(0);
      }
      else if ( pid > 0 ) { // Eseguito dal padre
         printf("%d: child created with PID %d\n", getpid(), pid);
      }
      else {
         perror("Fork error:");
         exit(1);
      }
   }
}
```

i = 2

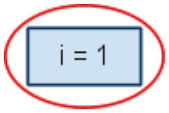
Processo P1

```
void main(int argc, char* argv[]) {
   int i, j, k, pid, status, n_children;
   char term;
   n_children = atoi(argv[1]);
   term = argv[2][0];
   for ( i=0; i<n_children; i++ ) {
      pid = fork();
      if ( pid == 0 ) { // Eseguito dai figli
         if ( term == '1' )
            exit(0);
      }
      else if ( pid > 0 ) { // Eseguito dal padre
         printf("%d: child created with PID %d\n", getpid(), pid);
      }
      else {
         perror("Fork error:");
         exit(1);
      }
   }
}
```

i = 0

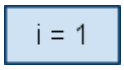
Processo P1

```
void main(int argc, char* argv[]) {
   int i, j, k, pid, status, n_children;
   char term;
   n_children = atoi(argv[1]);
   term = argv[2][0];
   for ( i=0; i<n_children; i++ ) {
      pid = fork();
      if ( pid == 0 ) { // Eseguito dai figli
         if ( term == '1' )
            exit(0);
      }
      else if ( pid > 0 ) { // Eseguito dal padre
         printf("%d: child created with PID %d\n", getpid(), pid);
      }
      else {
         perror("Fork error:");
         exit(1);
      }
   }
}
```

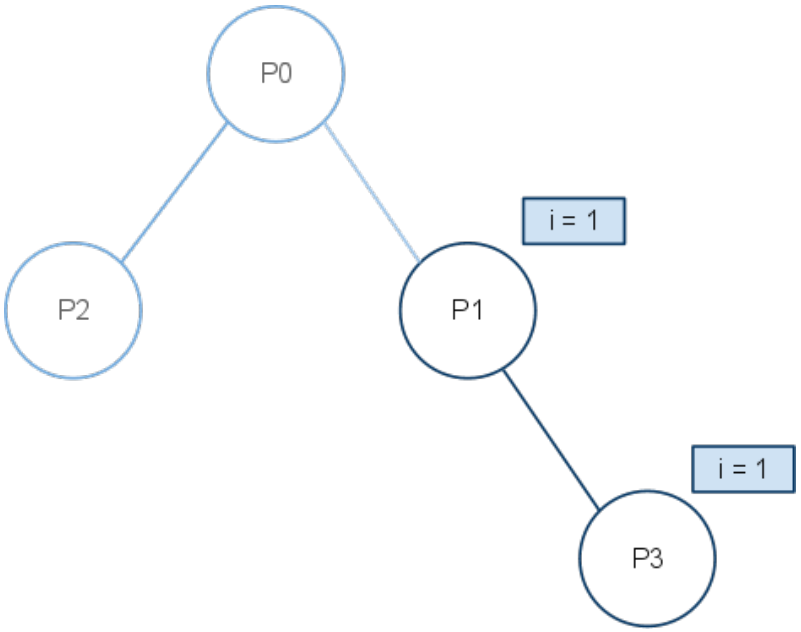


Processo P1

```
void main(int argc, char* argv[]) {
   int i, j, k, pid, status, n_children;
   char term;
   n_children = atoi(argv[1]);
   term = argv[2][0];
   for ( i=0; i<n_children; i++ ) {
      pid = fork();
      if ( pid == 0 ) { // Eseguito dai figli
         if ( term == '1' )
            exit(0);
      }
      else if ( pid > 0 ) { // Eseguito dal padre
         printf("%d: child created with PID %d\n", getpid(), pid);
      }
      else {
         perror("Fork error:");
         exit(1);
      }
   }
}
```



Simulazione di esecuzione (7/7)



Morale

- Quando si la *system call* **fork()**, bisogna sempre tener presente che i dati del padre vengono duplicati nel figlio
- Trascurare questo "dettaglio" può portare a comportamenti indesiderati

Esercizio 1

Scrivere un programma C con la seguente interfaccia

./compila_esegui <file1.c> <file2.c> ... <fileN.c>

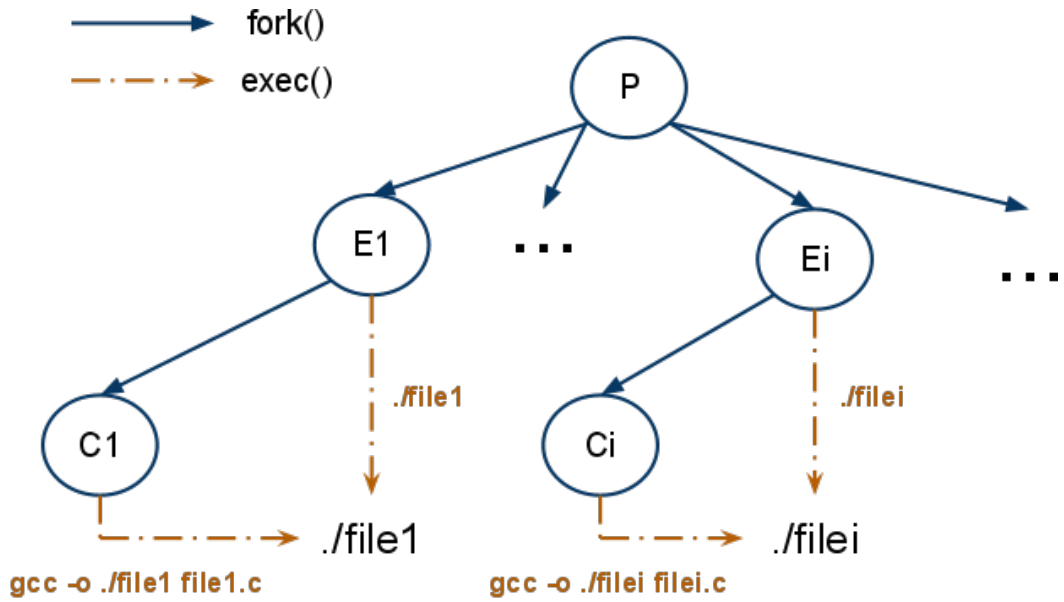
- <file1.c>, ..., <fileN.c> sono sorgenti C
- Il processo padre genera 2*N processi figli e/o nipoti
  - 2 processi per ogni sorgente passato come argomento
  - per ogni sorgente, un processo si occuperà di compilare il file
  - un altro (**distinto**) si occuperà di mettere in esecuzione il file eseguibile risultante
- Si generino i processi figli sequenzializzando il meno possibile le operazioni di compilazione ed esecuzione

Vincoli di sincronizzazione

- Tutti i processi di **compilazione** possono essere messi in esecuzione in maniera ***concorrente***
- La compilazione deve avvenire prima dell'esecuzione
  - Il processo che esegue deve ***sincronizzarsi*** col processo che compila

- Il processo esecutore **attende** il *termine dell'esecuzione* del processo compilatore
  - Relazione di **gerarchia** tra processo esecutore e processo compilatore

Schema di generazione



Schema di soluzione (1/2)

```
int main (int argc, char* argv[]) {
   int num_files, i;
   /* ... controllo argomenti ... */
   num_files = argc - 1;
   for( i=0; i<num_files; i++ ) {
      int pid = fork();
      if ( pid > 0 ) { /*...operazioni processo padre...*/ }
      else if ( pid == 0 ) { /*Processo figlio*/
        /*...recupero nome sorgente ed eseguibile...*/
         pid = fork();
         if ( pid > 0 ) { /*Processo figlio */
            int status, terminated_pid;
            terminated_pid = wait(&status);
            /*...verifica di condizioni di errore...*/
            if ( WEXITSTATUS(status) == 0 ) {
               execl( executable, executable, (char*)0 );
               perror(/*...error msg...*/);
               exit(1);
            }
/*...Controll eventuale stato d'errore...*/
```

CONTINUA...

Schema di soluzione (2/2)

... CONTINUA

```
            }
            else if ( pid == 0 ) { /* Processo nipote */
               execl ("/usr/bin/gcc", "/usr/bin/gcc", "-o",
                  executable, source_file, (char*)0 );
               perror( /* ... error msg ...*/ );
               exit(1);
            }
            /* ... controllo errori fork ...*/
         }
         /*...controllo errori fork...*/
      }
      for ( i=0; i<num_files; i++ ) {
         int wp, status;
         wp = wait(&status);
         /*...controllo errori...*/
      }
      return 0;
}
```

Schema di soluzione (2/2)

... CONTINUA

```
            }
            else if ( pid == 0 ) { /* Processo nipote */
               execl ("/usr/bin/gcc", "/usr/bin/gcc", "-o",
                  executable, source_file, (char*)0 );
               perror( /* ... error msg ...*/ );
               exit(1);
            }
            /* ... controllo errori fork ...*/
         }
         /*...controllo errori fork...*/
      }
      for ( i=0; i<num_files; i++ ) {
         int wp, status;
         wp = wait(&status);
         /*...controllo errori...*/
      }
      return 0;
}
```


1. quali processi eseguono questo codice?
2. a cosa serve?
3. perche` e` al di fuori del ciclo for di generazione?

Esercizio 2

Si realizzi un programma che, utilizzando le system call di Unix, soddisfi le seguenti specifiche.

- Interfaccia di invocazione:
  esegui_comando K COM1 COM2 ... COM
- Significato degli argomenti:
  - COM1, COM2, ..., COMN sono **N** stringhe che rappresentano il nome di un file (per semplicita`, si supponga che il direttorio di appartenenza dei ogni file COM sia nel **PATH**)
  - **K** e` un valore intero positivo (**strettamente minore** di N)

Esercizio 2 - Specifiche

Il processo iniziale (P0) deve mettere in esecuzione gli N comandi passati come argomenti, secondo la seguente logica:

- I primi **K** comandi passati come argomenti dovranno essere eseguiti **in parallelo** da **altrettanti figli** di P0.
- **al termine** dei primi K processi, i restanti **N-K** comandi dovranno essere eseguiti in sequenza da **altrettanti figli e/o nipoti** di P0

Esercizio per casa

Un processo padre P0 generi N figli.

- Consideriamo due possibili gerarchie:
  1. **N processi fratelli** (gerarchia a 2 liv. padre e N fratelli)
  2. **N processi, ciascuno figlio del precedente** (N+1 livelli: padre, figlio, nipote, pronipote, ecc.)

- Si vuole mantenere una struttura dati globale per tenere traccia dei PID di tutti i processi (padre + figli)

int pid [N+1];

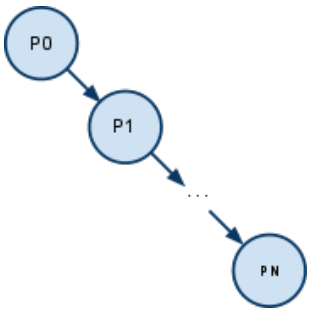
- Accio`, il risultato di ciascuna fork e` salvato nell'array, ad esempio:

```
int i = 0;
pid[0] = getpid(); // pid del padre
for ( i=0; i<N; i++) {
    pid[i] = fork();
    if (pid[i] == 0) { ... }
    else { ... }
}
```

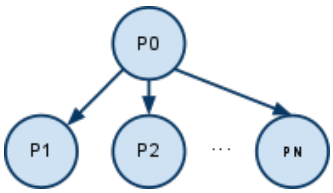
Esercizio per casa

Struttura **int pid[N]:**

- Con quali valori viene riempita?
- E` uguale per tutti i processi?



	pid[0]	pid[1]	pid[2]	...	pid[N]
P0					
P1					
P2					
...					
PN					



	pid[0]	pid[1]	pid[2]	...	pid[N]
P0					
P1					
P2					
...					
PN					