Terza Esercitazione

Unix – System Call Exec

Java – Introduzione Thread

Stefano Monti
smonti@deis.unibo.it

Unix - Esercizio 1

Scrivere un programma C con la seguente interfaccia:

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

dove file1.c,...., fileN.c sono file sorgenti C.

Il processo padre deve **generare 2*N processi** (figli e/o nipoti), 2 per ciascun sorgente; per ogni file, uno dei processi figli/nipoti si incaricherà di compilare il file, un altro processo figlio/nipote (DISTINTO dal precedente) di metterne in esecuzione l'eseguibile risultante.

Si generino i processi figli sequenzializzando il meno possibile le operazioni di compilazione ed esecuzione.

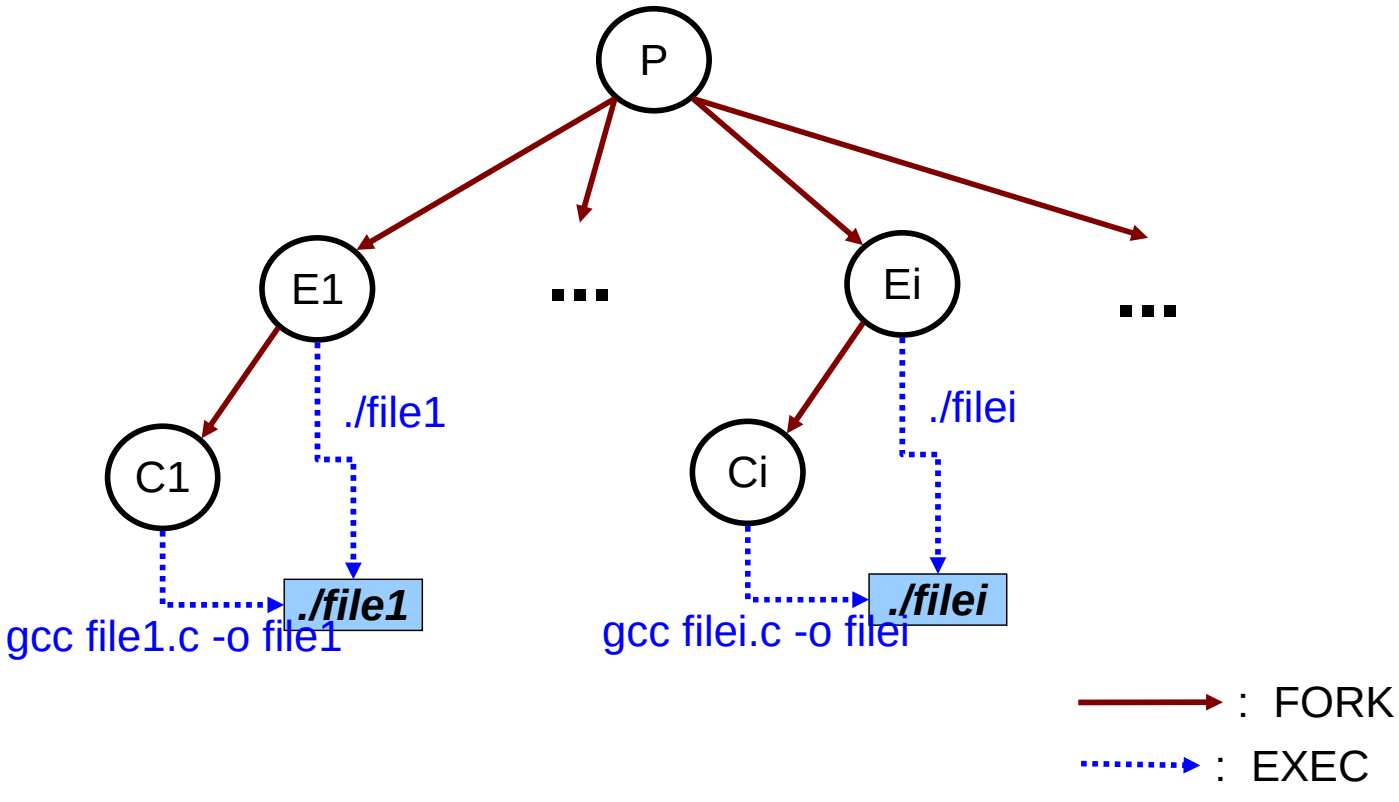
Vincoli di sincronizzazione

- I processi **compilatori** possono essere messi in esecuzione in maniera **concorrente**, ma...
- La **compilazione deve avvenire prima dell'esecuzione** --> il processo che esegue deve sincronizzarsi col processo che compila

con le primitive di sincronizzazione finora viste

- il processo esecutore ATTENDE il termine dell'esecuzione del processo compilatore --> relazione di gerarchia

Schema di generazione



```
for ( i=1; i<argc;i++){
        pid = fork();
        if (pid == 0){ /* figlio i-esimo*/
                pid = fork();
                if (pid == 0){ /* nipote i-esimo : compilazione*/
                        printf("Nipote: compilazione %s\n",argv[i]);
                        execl("/usr/bin/gcc", "gcc", argv[i], "-o", executableName, (char *)0);
                        perror("Errore in execl\n");
                           exit(1); }
                else if (pid > 0){ /* figlio i-esimo : esecuzione*/
                        printf("Figlio: esecuzione %s\n",executableName);
                        wait(&status); //attesa terminazione nipote
                        execl(executableName, executableName,  (char *)0);
                        perror("Errore in execl\n");
                           exit(1); }
                else{ perror("Errore in fork\n"); exit(1);}
                exit(0);}
        else if (pid > 0){ /* padre */
                 printf("Padre ....\n");
        }
        else {perror("Errore in fork\n"); exit(1); }
}
for ( i=1; i<argc;i++){
        wait(&status);
}
```

- quali processi eseguono questa porzione di codice?
- a cosa serve?
- perché è al di fuori del ciclo **for** di generazione?

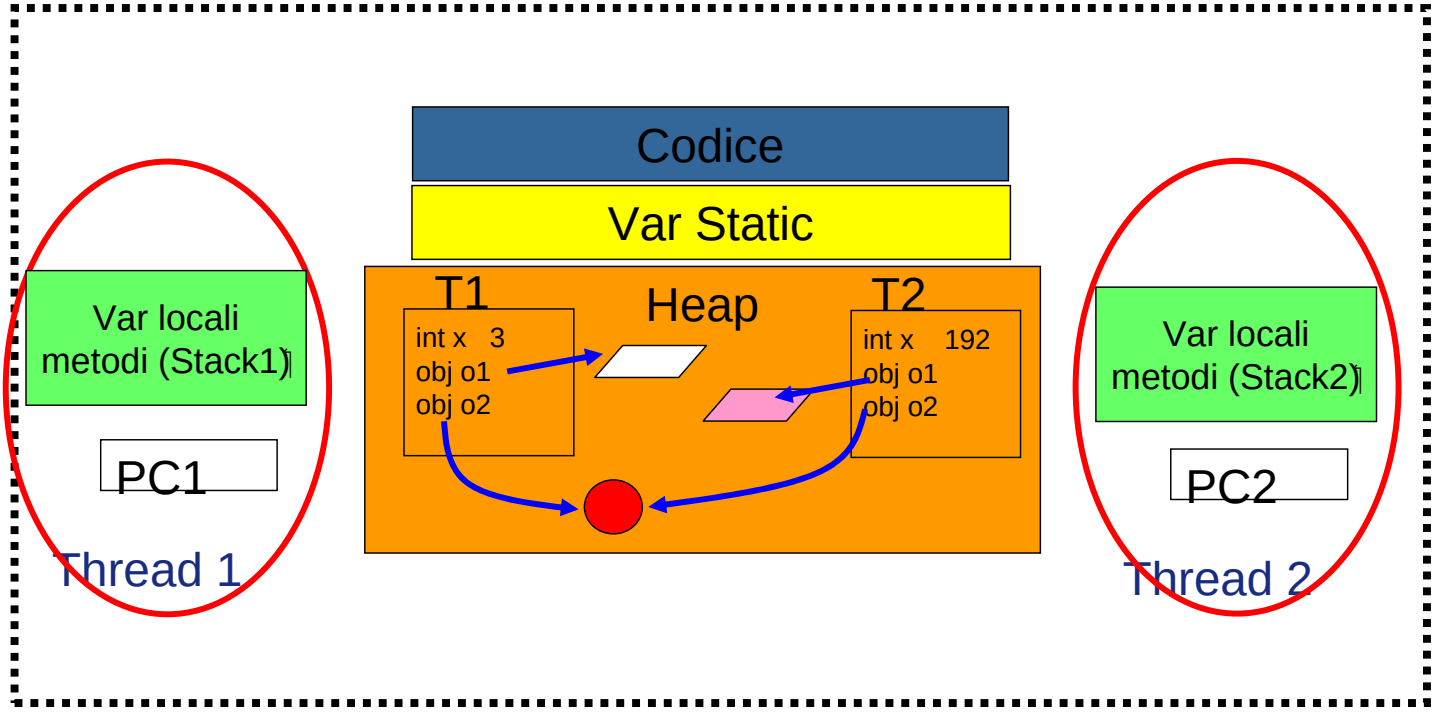
Estensione

- Come fa un esecutore a sapere se il processo compilatore ha compilato correttamente? In particolare, come può l'esecutore venire a conoscenza di eventuali errori di compilazione?

Interazione tra processi:

- modello ad **ambiente locale**: scambio di messaggi; primitive UNIX (non ancora viste)
- modello ad **ambiente globale**: cooperazione tramite accesso a **variabili condivise** tra processi (possibile in Java, non possibile in Unix)

Java - Modello ad ambiente globale



Accesso esclusivo

- Il modificatore **`synchronized`** permette di ***rendere esclusivo l'accesso di un thread ad un metodo***

- In assenza di **`synchronized`** accessi contemporanei da parte di più thread ***potrebbero*** portare l'oggetto condiviso in uno ***stato inconsistente***

Esercizio 1

- Due persone possono prelevare soldi da uno ***stesso conto***
- Le operazioni di prelievo possono essere autorizzate solo ***finchè il saldo del conto è positivo***
- Purtoppo i ritardi del terminale di accesso impongono che gli utenti attendano 3 secondi tra il test sulla condizione e l'aggiornamento del saldo

Esercizio 1

- Il saldo iniziale è pari a 10000 €
- Le due persone tentano continuamente di prelevare 3000 €

Suggerimenti:

- Si includa nel main() una verifica del saldo del conto ogni 2 secondi
- Si modelli il conto come classe autonoma e l'operazione di prelievo come un suo singolo metodo
- Si noti cosa succede se si permette ***un accesso simultaneo*** al conto da parte delle ***due persone***

Classe TestMonitor

```
public static void main(String[] args){
  SimpleBankAccount b=new
     SimpleBankAccount(10000);
    Persona p1=new Persona(b);
    Persona p2=new Persona(b);
    p1.start();p2.start();
    for(int i=0;i<100;i++){
     System.out.println("Current Balance"
          +b.getBalance());
     try{Thread.sleep(2000);}
     catch(InterruptedException e){}
    }
}
```

Classe Persona

```
public class Persona extends Thread{
    public SimpleBankAccount b;
    public Persona(SimpleBankAccount b)
          {this.b=b;}
  public void run(){
    for (int x=0;x<3;x++){
          b.withdraw(3000);
       }
    }
}
```

Classe SimpleBankAccount

```
public class SimpleBankAccount {
    private int balance;
    public SimpleBankAccount(int initial)
     {balance=initial;}
  synchronized  public void withdraw(int amount){
     if(balance-amount>0){
           try{Thread.sleep(3000);} //Per evidenziare l'interferenza
           catch(InterruptedException e){}
           balance-=amount;
     }
    }
    public int getBalance(){return balance;}
}
```

Sospensione di un thread

`sleep(long millisec)`

- **sospende** momentaneamente il **thread corrente** per il numero di millisecondi specificati
- si tratta di un **metodo statico**, quindi va invocato sulla classe Thread
- può sollevare una **InterruptedException** che deve essere intercettata

Sincronizzazione

`wait()`- il thread che la invoca

- si ***blocca*** in attesa che un altro thread invochi **`notify()`** o **`notifyAll()`** per quell'oggetto
- deve essere ***in possesso del lock sull'oggetto***
- ***al momento della invocazione rilascia il lock***

Sincronizzazione

- **`notify()`** - il thread che la invoca

  risveglia uno dei thread in attesa, scelto **arbitrariamente**
- **`notifyAll()`** - il thread che la invoca

  risveglia tutti i thread in attesa: essi competeranno per l'accesso all'oggetto

- **`notifyAll()`** ***è preferibile*** ***(e può essere necessaria)*** se più thread possono essere in attesa

Esercizio 2

- Un padre decide di aprire un conto corrente per i suoi due figli
- Il primo figlio tenta di prelevare 100€ 1000 volte, il secondo 200€ 1000 volte
- Tali ***prelievi sono permessi*** solo se consentono di mantenere il ***saldo maggiore o uguale a 0***
- Il padre aggiunge 200€ se il conto ha saldo minore o uguale a 0 per 1500 volte

Esercizio 2

Suggerimenti:

- Si stampi il saldo del conto ogni volta che lo si modifica
- Si modellino il padre, i figli e il conto come classi separate

Si verifichi:

- Cosa accade se si utilizza **`notify()`** anziché **`notifyAll()`**
- Cosa accade se si utilizza **`notify()`** e si cambia l'ordine di attivazione dei thread