Introduzione alla programmazione concorrente

Nicola Corti & Michael Sanelli

GULP - Gruppo Utenti Linux Pisa

04/05/2011

La programmazione concorrente

Finora abbiamo scritto programmi che eseguono il loro codice sequenzialmente.

Ma la maggior parte dei programmi complessi che eseguono sul nostro sistema non esegue in modo sequenziale.

La maggior parte dei programmi complessi eseguono piú di un **processo contemporaneamente**.

Un sistema di questo tipo si chiama **Sistema Concorrente**

La programmazione concorrente

Finora abbiamo scritto programmi che eseguono il loro codice sequenzialmente.

Ma la maggior parte dei programmi complessi che eseguono sul nostro sistema non esegue in modo sequenziale.

La maggior parte dei programmi complessi eseguono piú di un **processo contemporaneamente**.

Un sistema di questo tipo si chiama **Sistema Concorrente**

La programmazione concorrente

Finora abbiamo scritto programmi che eseguono il loro codice sequenzialmente.

Ma la maggior parte dei programmi complessi che eseguono sul nostro sistema non esegue in modo sequenziale.

La maggior parte dei programmi complessi eseguono piú di un **processo contemporaneamente**.

Un sistema di questo tipo si chiama **Sistema Concorrente**

La programmazione concorrente

Finora abbiamo scritto programmi che eseguono il loro codice sequenzialmente.

Ma la maggior parte dei programmi complessi che eseguono sul nostro sistema non esegue in modo sequenziale.

La maggior parte dei programmi complessi eseguono piú di un **processo contemporaneamente**.

Un sistema di questo tipo si chiama **Sistema Concorrente**

Problematiche di un sistema concorrente

Un sistema concorrente presenta alcune problematiche:

- **Condivisione di risorse**: ci si deve assicurare che le risorse vengano accedute in mutua esclusione.
- **Stalli**: ci si deve assicurare che i processi non restino bloccati in attesa fra di loro su una risorsa.
- **Politiche**: dobbiamo operare delle scelte su quali processi far eseguire prima di altri, in modo da non penalizzare nessuno

Problematiche di un sistema concorrente

Un sistema concorrente presenta alcune problematiche:

- **Condivisione di risorse**: ci si deve assicurare che le risorse vengano accedute in mutua esclusione.
- **Stalli**: ci si deve assicurare che i processi non restino bloccati in attesa fra di loro su una risorsa.
- **Politiche**: dobbiamo operare delle scelte su quali processi far eseguire prima di altri, in modo da non penalizzare nessuno

Problematiche di un sistema concorrente

Un sistema concorrente presenta alcune problematiche:

- **Condivisione di risorse**: ci si deve assicurare che le risorse vengano accedute in mutua esclusione.
- **Stalli**: ci si deve assicurare che i processi non restino bloccati in attesa fra di loro su una risorsa.
- **Politiche**: dobbiamo operare delle scelte su quali processi far eseguire prima di altri, in modo da non penalizzare nessuno

Problematiche di un sistema concorrente

Un sistema concorrente presenta alcune problematiche:

- **Condivisione di risorse**: ci si deve assicurare che le risorse vengano accedute in mutua esclusione.
- **Stalli**: ci si deve assicurare che i processi non restino bloccati in attesa fra di loro su una risorsa.
- **Politiche**: dobbiamo operare delle scelte su quali processi far eseguire prima di altri, in modo da non penalizzare nessuno

Problematiche di un sistema concorrente

Un sistema concorrente presenta alcune problematiche:

- **Condivisione di risorse**: ci si deve assicurare che le risorse vengano accedute in mutua esclusione.
- **Stalli**: ci si deve assicurare che i processi non restino bloccati in attesa fra di loro su una risorsa.
- **Politiche**: dobbiamo operare delle scelte su quali processi far eseguire prima di altri, in modo da non penalizzare nessuno

Problematiche legate alla comunicazione

Inoltre i processi hanno inoltre bisogno di comunicare:

- **Invio di messaggi**: per scambiare informazioni fra processi differenti.
- **Invio di segnali**: per sincronizzarsi nell'eseguire un'operazione in modo cooperativo.

Problematiche legate alla comunicazione

Inoltre i processi hanno inoltre bisogno di comunicare:

- **Invio di messaggi**: per scambiare informazioni fra processi differenti.
- **Invio di segnali**: per sincronizzarsi nell'eseguire un'operazione in modo cooperativo.

Problematiche legate alla comunicazione

Inoltre i processi hanno inoltre bisogno di comunicare:

- **Invio di messaggi**: per scambiare informazioni fra processi differenti.
- **Invio di segnali**: per sincronizzarsi nell'eseguire un'operazione in modo cooperativo.

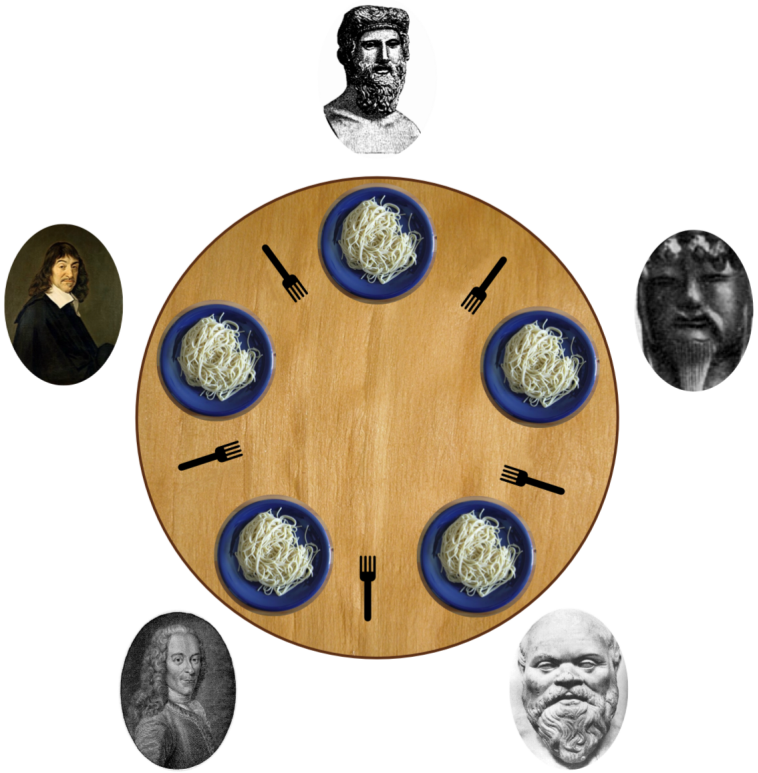
Problematiche legate alla comunicazione

Inoltre i processi hanno inoltre bisogno di comunicare:

- **Invio di messaggi**: per scambiare informazioni fra processi differenti.
- **Invio di segnali**: per sincronizzarsi nell'eseguire un'operazione in modo cooperativo.

I filosofi a cena

Un problema classico legato al concetto di concorrenza é quello dei *filosofi a cena*



I filosofi a cena

Abbiamo 5 filosofi e 5 forchette; disposti come nella figura precedente. Ogni filosofo alterna momenti in cui vuole mangiare a momenti in cui riflette, ma quando vuol mangiare deve aver disponibile entrambi le forchette alla sua destra, e alla sua sinistra.

Un probabile stallo...

Dobbiamo assicurarci che i filosofi non restino bloccati ognuno con una forchetta in mano, in attesa della seconda.

I filosofi a cena

Abbiamo 5 filosofi e 5 forchette; disposti come nella figura precedente. Ogni filosofo alterna momenti in cui vuole mangiare a momenti in cui riflette, ma quando vuol mangiare deve aver disponibile entrambi le forchette alla sua destra, e alla sua sinistra.

Un probabile stallo...

Dobbiamo assicurarci che i filosofi non restino bloccati ognuno con una forchetta in mano, in attesa della seconda.

I filosofi a cena

Abbiamo 5 filosofi e 5 forchette; disposti come nella figura precedente. Ogni filosofo alterna momenti in cui vuole mangiare a momenti in cui riflette, ma quando vuol mangiare deve aver disponibile entrambi le forchette alla sua destra, e alla sua sinistra.

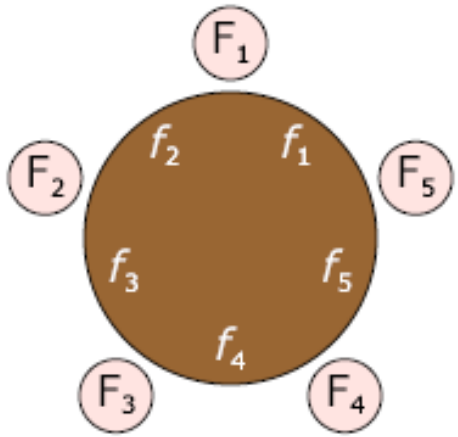
Un probabile stallo...

Dobbiamo assicurarci che i filosofi non restino bloccati ognuno con una forchetta in mano, in attesa della seconda.

Una possibile soluzione

Una possibile soluzione

Si potrebbe numerare le risorse, ed obbligare i filosofi a prendere le forchette in ordine crescente.



Cosí facendo si nota che il filosofo **5** blocca il possibile stallo, rendendo cosí impossibile la richiesta di risorse in modo ciclico.

Primo passo con la programmazione concorrente

Vediamo una breve codice per creare un processo figlio.

```c
#include <stdio.h>
#include <unistd.h>

int main(){

    int pid;

    pid = fork();
    if (pid == 0)
        printf("Sono il figlio :D\n");
    else
        printf("Sono il padre \n");
}
```

Primo passo con la programmazione concorrente

Vediamo una breve codice per creare un processo figlio.

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main(){
5
6      int pid;
7
8      pid = fork();
9      if (pid == 0)
10         printf("Sono il figlio :D\n");
11     else
12         printf("Sono il padre \n");
13 }
```

Primo passo con la programmazione concorrente

Vediamo una breve codice per creare un processo figlio.

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main(){
5
6      int pid;
7
8      pid = fork();
9      if (pid == 0)
10         printf("Sono il figlio :D\n");
11     else
12         printf("Sono il padre \n");
13 }
```

La fork

La funzione **fork** ci permette di creare un processo figlio del processo che la invoca.

Il valore di ritorno della fork puó essere:

- 0 Allora vuol dire che quel processo é il figlio
- > 0 Allora vuol dire che quel processo é il padre
- -1 Allora vuol dire che la fork é fallita

Useremo poi la funzione **execl** per far eseguire al figlio un altro programma.

La fork

La funzione **fork** ci permette di creare un processo figlio del processo che la invoca.

Il valore di ritorno della fork puó essere:

- 0 Allora vuol dire che quel processo é il figlio
- > 0 Allora vuol dire che quel processo é il padre
- -1 Allora vuol dire che la fork é fallita

Useremo poi la funzione **execl** per far eseguire al figlio un altro programma.

La fork

La funzione **fork** ci permette di creare un processo figlio del processo che la invoca.

Il valore di ritorno della fork puó essere:

- 0 Allora vuol dire che quel processo é il figlio
- > 0 Allora vuol dire che quel processo é il padre
- -1 Allora vuol dire che la fork é fallita

Useremo poi la funzione **execl** per far eseguire al figlio un altro programma.

La fork

La funzione **fork** ci permette di creare un processo figlio del processo che la invoca.

Il valore di ritorno della fork puó essere:

- 0 Allora vuol dire che quel processo é il figlio
- > 0 Allora vuol dire che quel processo é il padre
- -1 Allora vuol dire che la fork é fallita

Useremo poi la funzione **execl** per far eseguire al figlio un altro programma.

La fork

La funzione **fork** ci permette di creare un processo figlio del processo che la invoca.

Il valore di ritorno della fork puó essere:

- 0 Allora vuol dire che quel processo é il figlio
- > 0 Allora vuol dire che quel processo é il padre
- -1 Allora vuol dire che la fork é fallita

Useremo poi la funzione **execl** per far eseguire al figlio un altro programma.

La fork

La funzione **fork** ci permette di creare un processo figlio del processo che la invoca.

Il valore di ritorno della fork puó essere:

- 0 Allora vuol dire che quel processo é il figlio
- > 0 Allora vuol dire che quel processo é il padre
- -1 Allora vuol dire che la fork é fallita

Useremo poi la funzione **execl** per far eseguire al figlio un altro programma.

La fork

La funzione **fork** ci permette di creare un processo figlio del processo che la invoca.

Il valore di ritorno della fork puó essere:

- 0 Allora vuol dire che quel processo é il figlio
- > 0 Allora vuol dire che quel processo é il padre
- -1 Allora vuol dire che la fork é fallita

Useremo poi la funzione **execl** per far eseguire al figlio un altro programma.

Esempio di uso della execl

```c
#include <stdio.h>
#include <unistd.h>

int main(){

    int pid;

    pid = fork();
    if (pid == 0)
        {sleep(10);
        execl("/bin/ls", "ls", "-l", (char *)NULL);}
    else
        {printf("Sono il padre \n");
        sleep(20);}
}
```

Guardiamo come si evolve la gerarchia dei processi (su `top` o `gnome-system-monitor`).

Esempio di uso della execl

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main(){
5
6      int pid;
7
8      pid = fork();
9      if (pid == 0)
10         {sleep(10);
11         execl("/bin/ls", "ls", "-l", (char *)NULL);}
12     else
13         {printf("Sono il padre \n");
14         sleep(20);}
15 }
```

Guardiamo come si evolve la gerarchia dei processi (su `top` o `gnome-system-monitor`).

Esempio di uso della execl

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main(){
5
6      int pid;
7
8      pid = fork();
9      if (pid == 0)
10         {sleep(10);
11         execl("/bin/ls", "ls", "-l", (char *)NULL);}
12     else
13         {printf("Sono il padre \n");
14         sleep(20);}
15 }
```

Guardiamo come si evolve la gerarchia dei processi (su `top` o `gnome-system-monitor`).

Slides realizzate da:

Nicola Corti - corti.nico [at] gmail [dot] com

Michael Sanelli - michael [at] sanelli [dot] org

Slides realizzate con LaTeX Beamer, ed utilizzando interamente software libero.