Programmazione concorrente – Principi e introduzione ai processi

by Leonardo Giordani
<leo.giordani(at)libero.it>

About the author:

Studente presso la Facoltà di Ingegneria delle Telecomunicazioni del Politecnico di Milano, si occupa di amministrazione di rete e di programmazione (prevalentemente in Assembly e C/C++). Ha conosciuto la realtà di Linux/Unix nel 1999 e da allora si dedica quasi esclusivamente ad essa.

Abstract:

Questa serie di articoli vuole introdurre il lettore al principio di multitasking e alla sua implementazione nel sistema operativo Linux. Partendo dai concetti teorici che stanno alla base del multitasking si arriverà a scrivere una completa applicazione dimostrativa della comunicazione tra processi, con un semplice ma efficace protocollo di comunicazione.
Prerequisiti per la comprensibilità dell'articolo sono:

- Conoscenza minimale dell'uso della shell
- Conoscenza base del C (sintassi, cicli, librerie)

Tutti i riferimenti alle pagine di manuale sono posti tra parentesi dopo il nome del comando citato. Tutte le funzioni della glibc sono documentate tramite "info Libc".

Introduzione

Una delle svolte più importanti che si sono avute nella storia dei sistemi operativi è stata l'introduzione del concetto di multiprogrammazione, un metodo per interallacciare l'esecuzione di diversi programmi permettendo un utilizzo più intenso e costante delle risorse di sistema. Pensiamo anche solo ad una comune workstation, dove un utente può avere in esecuzione contemporaneamente un wordprocessor, un riproduttore audio, una coda di stampa, un navigatore web e chi più ne ha più ne metta, e ci renderemo subito conto dell'importanza pratica di questa tecnica. Come scopriremo più avanti, poi, questa piccola lista non è che una parte (minima) dell'insieme di programmi realmente in esecuzione sulla nostra macchina, nonostante sia la parte più "appariscente".

Il concetto di processo

Per poter effettuare l'interallacciamento dei programmi è necessario complicare notevolmente il sistema operativo; per poter evitare che ci siano conflitti tra i vari programmi è obbligatorio incapsulare ognuno di essi assieme a tutte le informazioni necessarie per la sua esecuzione.

Volendo dare un po' di nomenclatura tecnica diciamo che, dato un **PROGRAMMA** in esecuzione, in un certo istante è detto **CODICE** l'insieme delle istruzioni che lo compongono, **SPAZIO DI MEMORIA** l'attuale insieme di memoria della macchina occupato dai suoi dati e **STATO DEL PROCESSORE** l'insieme dei valori dei parametri del microprocessore, quali le flag o il Program Counter (l'indirizzo della prossima istruzione da eseguire).

Data una terna ci oggetti composta da CODICE, SPAZIO DI MEMORIA e STATO DEL PROCESSORE abbiamo univocamente definito un **PROGRAMMA IN ESECUZIONE**. Se quindi ad un certo momento durante il funzionamento della macchina salvassimo queste informazioni e le sostituissimo con altre relative ad un'altro programma il flusso di quest'ultimo proseguirebbe dal punto in cui era stato arrestato e noi avremmo realizzato l'interallacciamento di cui parlavamo prima. La terna di oggetti che abbiamo identificato ora è detta **PROCESSO** o **TASK**, concetto che quindi ha a che fare con quello di programma, ma che ne un'estensione.

Possiamo ora formalizzare ciò che accade alla workstation di cui parlavamo nell'introduzione: in ogni istante di tempo è in esecuzione un solo processo (c'è un solo microprocessore), e la macchina esegue parte del codice di quest'ultimo; dopo un certo istante di tempo molto breve detto QUANTO il processo in esecuzione viene sospeso e le sue informazioni sono salvate e sostituite con quelle relative ad un altro processo in attesa, il cui codice viene eseguito per un altro quanto di tempo e così via. Agli occhi dell'utente questo realizza la CONCORRENZA dei processi, ovvero l'esecuzione simultanea di più programmi.

L'introduzione del multitasking (o multiprogrammazione) crea un insieme di problemi non banali, come la gestione delle code dei processi in attesa (SCHEDULING), che però riguardano più strettamente l'architettura di un singolo sistema operativo. Non è escluso che in un futuro articolo si parli anche di questo, magari guardando più da vicino il codice del kernel.

I processi in Linux e Unix

Prima di avventurarci nella scrittura di codice multiprogrammato vediamo cosa possiamo scoprire riguardo ai processi che già sono in esecuzione sul nostro sistema; il comando che ci permette di avere queste informazioni è **ps(1)** che sta per "process status", stato dei processi. Aprendo una normale shell testuale e digitando il comando ps otterremo un output simile al seguente:

```
  PID TTY          TIME CMD
 2241 ttyp4    00:00:00 bash
 2346 ttyp4    00:00:00 ps
```

Premetto subito che questo elenco non è affatto completo, ma per adesso accontentiamoci di questo: ps ci ha restituito l'elenco di tutti i processi che sono stati lanciati dal terminale corrente. Vediamo subito che l'ultima colonna riporta il nome del comando che è stato dato per lanciare quel dato processo: ovviamente ps compare nell'elenco in quanto nell'istante in cui è stato lanciato è entrato nella lista dei processi in esecuzione. L'altro processo è la bash, la shell che sto usando sui miei terminali.

Tralasciamo per ora le informazioni TIME e TTY, e concentriamoci su PID, che sta per Process IDentifier, ovvero identificatore di processo. Il pid è un numero strettamente maggiore di zero che viene assegnato univocamente ad ogni processo che viene eseguito; chiaramente una volta terminato il processo il pid può

essere riutilizzato, ma abbiamo la garanzia che mentre il processo è in esecuzione il pid non verrà cambiato. Questo implica che l'output che ognuno di voi avrà dal comando ps sarà molto probabilmente diverso dal mio almeno per quanto riguarda i valori del pid. Per renderci effettivamente conto di questo proviamo ad aprire una nuova shell senza chiudere quella precedente: l'output del comando ps questa volta riporta il processo "bash" con il pid 2488 (o comunque differente da quello precedente), testimoniando che a discapito del nome si tratta di due processi differenti, nonostante il programma che eseguono sia uguale.

Vediamo ora come sia possibile ottenere una lista di tutti i processi attualmente in esecuzione sulla nostra Linux box. La pagina di manuale del comando ps (piuttosto corposa) riporta lo switch –e con la dizione "select all processes". Proviamo quindi a digitare "ps –e" ed otterremo una lista piuttosto lunga formattata in maniera analoga all'output precedentemente analizzato. Per analizzare più comodamente la lista salviamo l'output in un file, che chiameremo ps.log:

```
ps –e > ps.log
```

Adesso possiamo leggere con comodo il file utilizzando in nostro editor preferito (o semplicemente il comando less); come anticipato il numero di processi che sono in esecuzione è più alto di quanto ci aspettassimo. Notiamo infatti che in elenco non ci sono solamente applicazioni lanciate da noi nell'ambiente grafico, bensì un insieme di processi con nomi talora piuttosto oscuri: il numero e l'identità dei processi elencati dipende dalla configurazione del vostro sistema, ma possiamo notare alcune cose comuni. Innanzitutto tutti noteranno come il processo che ha il pid 1 sia **init(8)**: questo non è un caso, ma fa parte dell'architettura di Linux. Il processo init, infatti, è il padre di tutti i processi, per cui è il primo ad essere eseguito. Un'altra cosa che tutti noteranno è la presenza di molti processi il cui nome termina con una "d": sono i cosidetti "demoni", e sono tra i processi più importanti del sistema. Approfondiremo il discorso su init e i demoni in un articolo successivo, quindi per ora accontentiamoci di convivere con queste due entità.

Multitasking nella libc

Dopo aver compreso il concetto di processo ed aver visto velocemente quanto esso sia importante nel sistema operativo possiamo procedere oltre iniziando a scrivere codice multitasking; dalla banale esecuzione simultanea di processi ci sposteremo poi verso una nuova problematica: la comunicazione tra processi concorrenti e la loro temporizzazione; vedremo due soluzioni molto eleganti al problema, i messaggi ed i semafori, i secondi dei quali verranno poi approfonditi nella trattazione dei thread. Inizieremo quindi a gettare le basi per la scrittura di un'applicazione completa basata su questi concetti.

La libreria standard C (libc, implementata in Linux nella glibc del progetto GNU) utilizza i meccanismi di multitasking implementati nello Unix System V (d'ora in poi SysV), uno Unix commerciale capostipite di una delle due implementazioni più famose di Unix; l'altra è BSD.

Nella libc viene definito il tipo pid_t come un integer contenente un pid. D'ora in poi per contenere i pid dei processi utilizzeremo sempre questo tipo di variabile, per un puro scopo di chiarezza: utilizzare come variabile per i pid un int non presenta nessuna differenza.

Cominciamo a vedere quale funzione ci permette di conoscere il pid del processo che contiene il nostro programma

```
pid_t getpid (void)
```

(contenuta insieme a pid_t negli header unistd.h e sys/types.h) e scriviamo un programma che come scopo ha quello di stampare sullo standard output il suo pid. Con un qualsiasi editor scrivete il seguente codice

```
#include <unistd.h>
#include <sys/types.h>
```

```
#include <stdio.h>

int main()
{
  pid_t pid;

  pid = getpid();
  printf("Il pid assegnato al processo è %d\n", pid);

  return 0;
}
```

Salvate il programma come print_pid.c e compilatelo con

```
gcc -o print_pid print_pid.c
```

che creerà un eseguibile print_pid. Vi ricordo che se la directory corrente non è nel path è necessario eseguire il programma come "./print_pid". Eseguendo il programma non abbiamo particolari sorprese: esso stampa un numero maggiore di zero e, se eseguito più volte, è probabile che il numero sia progressivo; questo non è obbligatorio, dato che potrebbe esserci qualche altro processo che viene creato da un altro programma tra un esecuzione e l'altra di print_pid. Provate ad esempio ad eseguire ps tra due esecuzioni di print_pid...

Ora è tempo di imparare a creare un processo, ma è necessario spendere ancora due parole su quello che avviene realmente durante questa azione. Quando un programma (che è contenuto nel processo A) crea un altro processo B i due sono identici, ovvero contengono lo stesso codice, hanno la memoria piena degli stessi dati (non la stessa memoria, ma due copie identiche della stessa) e hanno il medesimo stato del processore. A questo punto i due si possono evolvere in maniere indipendenti, per esempio in dipendenza degli input dell'utente o di fattori casuali. Il processo A è detto "processo padre", mentre il B è il "processo figlio"; adesso possiamo capire meglio l'attributo di "padre di tutti i processi" dato a init. La funzione che crea un nuovo processo è

```
pid_t fork(void)
```

e deve il suo nome proprio all'azione di biforcare l'esecuzione del programma. Il dato che viene restituito è un pid, ma merita una considerazione molto particolare. Abbiamo detto che il processo attuale si duplica in padre e figlio, che si interlacceranno nell'esecuzione assieme agli altri processi del sistema, svolgendo due attività differenti; ma subito dopo la creazione sarà in esecuzione il processo padre o il processo figlio? Ebbene la risposta è semplice: uno dei due. La decisione di quale processo deve venire eseguito viene infatti presa da una parte di sistema operativo detta scheduler, il quale non bada al fatto che un processo sia padre o figlio, ma segue un proprio algoritmo indipendente basato su parametri prestazionali.

D'altronde è necessario sapere quale processo è in esecuzione, poiché visto che il codice è lo stesso, entrambi i processi conterranno il codice destinato al padre e quello destinato al figlio, ma ciascuno di essi eseguirà solo quello destinato a sé stesso. Per chiarire definitivamente il concetto seguiamo lo schema seguente:

```
- BIFORCAZIONE
- SE SEI IL FIGLIO ESEGUI (...)
- SE SEI IL PADRE ESEGUI (...)
```

che rappresenta in metalinguaggio il codice del nostro programma relativo alla biforcazione. Sveliamo quindi l'arcano: la funzione fork restituisce '0' al processo figlio e il pid del figlio al processo padre. In questo modo basterà che il programma testi se il pid restituito sia zero per sapere dentro quale processo è in esecuzione. Il codice necessario ad effettuare questa operazione è il seguente

```
int main()
{
```

```
 pid_t pid;

 pid = fork();
 if (pid == 0)
 {
  CODICE PROCESSO FIGLIO
 }
 CODICE PROCSSO PADRE
}
```

Vediamo allora il primo vero esempio di codice multitasking, che potete salvare in un file fork_demo.c e compilare come per il programma precedente. I numeri di riga sono stati ovviamente posti solamente per la comprensione del codice. Il programma si biforcherà in due ciascuno dei quali scriverà sullo schermo una stringa; eseguendolo ci aspettiamo di ottenere un'interallacciamento dei due output.

```
(01) #include <unistd.h>
(02) #include <sys/types.h>
(03) #include <stdio.h>

(04) int main()
(05) {
(05)   pid_t pid;
(06)   int i;

(07)   pid = fork();

(08)   if (pid == 0){
(09)     for (i = 0; i < 8; i++){
(10)       printf("-FIGLIO-\n");
(11)     }
(12)     return 0;
(13)   }

(14)   for (i = 0; i < 8; i++){
(15)     printf("+PADRE+\n");
(16)   }

(17)   return 0;
(18) }
```

Le righe (01)–(03) contengono l'inclusione delle librerie necessarie alle funzioni multitasking e all'I/O standard.
Il main (04), come sempre in Linux, restituisce un intero che normalmente è zero se il programma è terminato normalmente e un numero differente da zero per indicare un qualche codice di errore, possibilità che, per semplicità, abbiamo assunto non si possa verificare. Definiamo poi il tipo di dato che contiene un pid (05) e un intero che fungerà da contatore per i cicli (06). In realtà, come abbiamo già detto, questi due tipi sono identici, ma la loro differenziazione è utile a mantenere chiarezza.
Alla riga (07) chiamiamo la funzione fork che restituirà, come abbiamo già detto, un pid pari a zero nel programma eseguito nel processo figlio e il pid del processo figlio nel programma eseguito nel processo padre. Alla riga (08) testiamo proprio questo pid, così come abbiamo spiegato prima. A questo punto il codice fino alla parentesi di riga (13) verrà eseguito nel processo figlio, mentre il codice dalla riga (14) alla (17) verrà eseguito nel processo padre.
Il codice dalle righe dalla (09) alla (12) esegue un ciclo che stampa 8 volte la stringa "–FIGLIO–" e poi termina restituendo 0. Questo è molto importante, poiché senza la riga (12) il figlio, una volta eseguito il ciclo, proseguirebbe nell'esecuzione del ciclo successivo (quello destinato al padre); errori di questo tipo sono estremamente facili da compiere quando si scrive codice multitasking, ma altrettanto difficili da trovare, visto che l'esecuzione di un programma di questo tipo (soprattutto se complesso) porta a risultati differenti ad ogni esecuzione, rendendo il debugging basato sui risultati quasi impossibile.

Le righe dalla (14) alla (17) contengono il ciclo del processo padre, che termina anch'esso con un "return 0".

Eseguendo il programma forse resteremo delusi: non posso assicurarvi che il risultato sia una vera e propria mescolanza delle due stringhe, e questo a causa della velocità di esecuzione di un ciclo così breve come quello da noi imposto; è molto probabile che ciò che otterremo sarà una serie di stringhe "+PADRE+" seguite da una serie di stringhe "–FIGLIO–" o il contrario. Provate comunque ad eseguire più volte il programma e il risultato potrebbe cambiare, anche se di poco.

Volendo avere un riscontro più visuale del multitasking della nostra applicazione potremmo pensare di inserire un ritardo casuale prima di ogni chiamata a printf, mediante la funzione sleep e la funzione rand

```
sleep (rand()%4)
```

questo obbliga il programma ad attendere un numero casuale di secondi tra 0 e 3 (il segno percentuale restituisce il resto della divisione intera). Modifichiamo quindi i cicli inserendo questa riga prima della (10) e della (15)

```
(09)  for (i = 0; i < 8; i++){
(->)    sleep (rand()%4);
(10)    printf("-FIGLIO-\n");
(11)  }
```

A questo punto salviamo come fork_demo2.c, compiliamo ed eseguiamo. A parte la lentezza notiamo anche una certa differenza nell'ordine delle stringhe stampate a video, e una maggiore differenza da un'esecuzione all'altra. Ciò che ottengo io, ad esempio, è

```
[leo@mobile ipc2]$ ./fork_demo2
-FIGLIO-
+PADRE+
+PADRE+
-FIGLIO-
-FIGLIO-
+PADRE+
+PADRE+
-FIGLIO-
-FIGLIO-
+PADRE+
+PADRE+
-FIGLIO-
-FIGLIO-
-FIGLIO-
+PADRE+
+PADRE+
[leo@mobile ipc2]$
```

Per terminare l'articolo diamo una scorsa alle problematiche che ci si presentano di fronte ora: abbiamo la possibilità di creare un certo numero di processi figli dato un processo padre, in modo che essi eseguano delle operazioni differenti da quelle del padre stesso in modo concorrente; ma il processo padre spesso ha la necessità di poter comunicare con i processi figli, o almeno di sincronizzarsi con essi, in modo che le operazioni vengano eseguite al momento opportuno. Ebbene vediamo subito un primo modo di ottenere questa sincronizzazione tra processi mediante la funzione di attesa

```
pid_t waitpid (pid_t PID, int *STATUS_PTR, int OPTIONS)
```

dove PID è il pid del processo di cui attendiamo il completamento, STATUS_PTR un puntatore ad un intero che conterrà lo stato del processo figlio (viene posto a NULL se l'informazione non serve) e OPTIONS un insieme di opzioni che per ora possiamo trascurare. Scriviamo quindi un esempio di programma in cui il padre

crea il processo figlio e ne attende il completamento

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

int main()
{
  pid_t pid;
  int i;

  pid = fork();

  if (pid == 0){
    for (i = 0; i < 14; i++){
      sleep (rand()%4);
      printf("-FIGLIO-\n");
    }
    return 0;
  }

  sleep (rand()%4);
  printf("+PADRE+ Attendo il completamento del processo figlio...\n");
  waitpid (pid, NULL, 0);
  printf("+PADRE+ ...terminato\n");

  return 0;
}
```

Lo sleep nel codice del padre è stato inserito per differenziare l'esecuzione. Salviamo il codice come fork_demo3.c compiliamo ed eseguiamo. Abbiamo appena programmato la nostra prima applicazione multitasking sincronizzata!

Nel prossimo articolo parleremo più diffusamente della sincronizzazione e della comunicazione tra processi cominciando a gettare le basi per la nostra applicazione dimostrativa; per ora vi invito a scrivere dei programmi che sfruttino le funzioni descritte, e ad inviarmeli di modo che io possa utilizzarne alcuni per evidenziare soluzioni ingegnose o errori da evitare. Vi prego di inviarmi sia il file .c con il codice (commentato!) che un piccolo file di testo contenente una breve descrizione del programma, il vostro nome e il vostro indirizzo di posta elettronica. Buon lavoro!

Letture raccomandate

- Silberschatz, Galvin, Gagne, **Operating System Concepts – Sixth Edition**, Wiley&Sons, 2001
- Tanenbaum, WoodHull, **Operating Systems: Design and Implementation – Second Edition**, Prentice Hall, 2000
- Stallings, **Operating Systems – Fourth Edition**, Prentice Hall, 2002
- Bovet, Cesati, **Understanding the Linux Kernel**, O'Reilly, 2000

Per segnalazioni o domande vi invito a scrivermi all'indirizzo leo.giordani@libero.it