Sistemi Operativi[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

a.a. 2013/14

Sistemi Operativi

Bruschi Monga Re

Sincronizzazione con monitor pthreads

Shell
Shell programming
Esercizi
I/O
Esercizi
Tabella riassuntiva

Lezione XV: Concorrenza e sincronizzazione

POSIX threads

Sistemi Operativi

Bruschi Monga Re

Sincronizzazione con monitor pthreads

Shell
Shell programming
Esercizi
I/O
Esercizi
Tabella riassuntiva

Gli esempi visti finora usano clone per creare i thread, che però è una system call specifica di Linux. Lo standard POSIX specifica una serie di API per la programmazione concorrente chiamate pthread (su Linux saranno implementate tramite clone).

- "multiparadigma": ci concentriamo sul modello a monitor, con mutex e condition variable. (Nota: i monitor sono costrutti specifici nel linguaggio, pthread usa il C, quindi p.es. l'incapsulamento dei dati va curato a mano)

```
1  pthread_create(thread,attr,start_routine,arg)
2  pthread_exit (status)
3  pthread_join (threadid,status)
4  pthread_mutex_init (mutex,attr)
5  pthread_mutex_lock (mutex)
6  pthread_mutex_unlock (mutex)
7  pthread_cond_init (condition,attr)
8  pthread_cond_wait (condition,mutex)
9  pthread_cond_signal (condition)
10 pthread_cond_broadcast (condition)
```


Il pattern di base

Sistemi Operativi

Bruschi Monga Re

Sincronizzazione con monitor pthreads

Shell
Shell programming
Esercizi
I/O
Esercizi
Tabella riassuntiva

Tralasciando le inizializzazioni dei puntatori mutex e condition:

```
1   // T1
2   pthread_mutex_lock(mutex); // Acquisire il lock
3   while (!predicate) // fintantoché la condizione è falsa
4       pthread_cond_wait(condition, mutex); // block
5   pthread_mutex_unlock(mutex); // rilasciare il lock
6
7   // T2
8   // qualche thread rende vero il predicato cosí
9   pthread_mutex_lock(mutex); // Acquisire il lock
10  predicate = TRUE;
11  pthread_cond_signal(condition); // e lo segnala
12  pthread_mutex_unlock(mutex); // rilasciare il lock
```

Perché il mutex?

Il mutex è necessario per sincronizzare il controllo della condizione, altrimenti

```
1 // T1
2 pthread_mutex_lock(mutex);
3 while (!predicate)
4   //
5   //
6   pthread_cond_wait(condition, mutex);
7 pthread_mutex_unlock(mutex);
```

```
1   // T2
2   //
3   //
4   predicate = TRUE;
5   pthread_cond_signal(condition);
6   //
7   //
```


Produttore e consumatore

- Il produttore smette di produrre se il buffer è pieno e deve essere avvisato quando non lo è piú (può ricominciare a produrre)
- Il consumatore smette di consumare se il buffer è vuoto e deve essere avvisato quando non lo è piú (può ricominciare a consumare)
- 2 condition variable: buffer pieno e buffer vuoto (ne servono due perché pieno $\neq \neg$ vuoto)

Lezione XVII: Unix power tools

Pipe

ls | sort

```
1  int main(void){
2          int fd[2], nbytes; pid_t childpid;
3          char string[] = "Hello, world!\n";
4          char readbuffer[80];
5
6          pipe(fd);
7          if(fork() == 0){
8                  /* Child process closes up input side of pipe */
9                  close(fd[0]);
10                 write(fd[1], string, (strlen(string)+1));
11                 exit(0);
12         } else {
13                 /* Parent process closes up output side of pipe */
14                 close(fd[1]);
15                 nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
16                 printf("Received string: %s", readbuffer);
17         }
18         return(0);}
```

Pipe (cont.)

```
1        if(fork() == 0)
2        {
3                /* Close up standard input of the child */
4                close(0);
5
6                /* Duplicate the input side of pipe to stdin */
7                dup(fd[0]);
8                execlp("sort", "sort", NULL);
9        }
```


Un vero linguaggio di programmazione

La shell è un vero (Turing-completo) linguaggio di programmazione (interpretato)

- Variabili (create al primo assegnamento, uso con `$`, export in un'altra shell).
  - x="ciao"; y=2 ; /bin/echo "$x $y $x"
- Istruzioni condizionali (valore di ritorno 0 $\rightsquigarrow$ true)
  - **if** /bin/ls piripacchio; then /bin/echo ciao; **else** /bin/echo buonasera; **fi**
- Iterazioni su insiemi
  - **for** i in a b c d e; **do** /bin/echo $i; done
- Cicli
  - /usr/bin/touch piripacchio
  2 **while** /bin/ls piripacchio; **do**
  3 /usr/bin/sleep 2
  4 /bin/echo ciao
  5 done & (/usr/bin/sleep 10 ; /bin/rm piripacchio)

Esercizi

1. Per ciascuno dei file `dog`, `cat`, `fish` controllare se esistono nella directory `bin` (hint: usare `/bin/ls` e nel caso scrivere `‘‘Trovato’’`)
2. Consultare il manuale (programma `/usr/bin/man`) del programma `/bin/test` (per il manuale man test)
3. Riscrivere il primo esercizio facendo uso di `test`

Input e Output

In generale il paradigma UNIX permette alle applicazioni di fare I/O tramite:

Input

- Parametri al momento del lancio
- Variabili *d'ambiente*
- File (tutto ciò che può essere gestito con le syscall `open, read, write, close`)
  - Terminale (interfaccia testuale)
  - Device (per es. il mouse potrebbe essere `/dev/mouse`)
  - Rete (socket)

Output

- Valore di ritorno
- Variabili *d'ambiente*
- File (tutto ciò che può essere gestito con le syscall `open, read, write, close`)
  - Terminale (interfaccia testuale)
  - Device (per es. lo schermo in modalità grafica potrebbe essere `/dev/fb`)
  - Rete (socket)

Redirezioni

Ad ogni processo sono sempre associati tre file (già aperti)

- Standard input (Terminale, tastiera)
- Standard output (Terminale, video)
- Standard error (Terminale, video, usato per le segnalazione d'errore)

Possono essere *rediretti*

- /usr/bin/sort < lista Lo stdin è il file `lista`
- /bin/ls > lista Lo stdout è il file `lista`
- /bin/ls piripacchio 2> lista Lo stderr è il file `lista`
- (echo ciao & date ; ls piripacchio) 2> errori 1>output

Pipe

La pipe è un canale, analogo ad un file, bufferizzato in cui un processo scrive e un altro legge. Con la shell è possibile collegare due processi tramite una pipe anonima.
Lo stdout del primo diventa lo stdin del secondo
/bin/ls | sort
ls −lR / | sort | more
funzionalmente equivalente a
ls −lR >tmp1; sort <tmp1 >tmp2; more<tmp2; rm tmp∗
Molti programmi copiano lo stdin su stdout dopo averlo elaborato: sono detti filtri.

Command substitution

Con una pipe è possibile "collegare" lo stdout di un programma con lo stdin di un altro.
Per usare l'output di un programma sulla riga di comando di un altro programma, occorre usare la command substitution
/bin/ls −l $(/usr/bin/which sort)

Esercizi

1. Verificare qual è il valore di ritorno di una *pipeline*, anche in caso che qualcuno dei "filtri" fallisca.
2. Scrivere una *pipeline* di comandi che identifichi il le informazioni sul processo `dropbear` (`ps, grep`)
3. Scrivere una *pipeline* di comandi che identifichi il solo processo con il PPID piú alto (`ps, sort, tail`)
4. Ottenere il numero totale dei file contenuti nelle directory `/usr/bin` e `/var` (`ls, wc, expr`)
5. Si immagini di avere un file contenente il sorgente di un programma scritto in un linguaggio di programmazione in cui i commenti occupino intere righe che iniziano con il carattere #. Scrivere una serie di comandi per ottenere il programma senza commenti. (`grep`)
6. Ottenere la somma delle occupazioni dei file delle directory `/usr/bin` e `/var` (`du, cut`)

Tabella riassuntiva

Prog. (sez. man)	Descrizione
`ls (1)`	list directory contents
`echo (1)`	display a line of text
`touch (1)`	change file timestamps
`sleep (1)`	delay for a specified amount of time
`rm (1)`	remove files or directories
`cat (1)`	concatenate files and print on the standard output
`man (1)`	an interface to the on-line reference manuals
`test (1)`	check file types and compare values
`sort (1)`	sort lines of text files
`date (1)`	print or set the system date and time
`less (1)`	file perusal filter for crt viewing
`which (1)`	locate a command
`ps (1)`	report a snapshot of the current processes.
`tail (1)`	output the last part of files
`wc (1)`	print the number of newlines, words, and bytes in files
`dc (1)`	An arbitrary precision calculator language
`grep (1)`	print lines matching a pattern
`cut (1)`	remove sections from each line of files
`du (1)`	print disk usage

Link

- "A Brief Introduction to Unix (With Emphasis on the Unix Philosophy)", Corey Satten `http://staff.washington.edu/corey/unix-intro.pdf`
- `http://en.wikipedia.org/wiki/Unix_philosophy`
- "The UNIX Time-Sharing System", Ritchie; Thompson `http://www.cs.berkeley.edu/~brewer/cs262/unix.pdf`