Introduzione al sistema multiprogrammato

G. Lettieri

2 Maggio 2017

1 Introduzione generale

Cominciamo a studiare come utilizzare i meccanismi hardware introdotti finora per realizzare un sistema in grado di eseguire più istanze di programmi (processi) concorrentemente.

Il sistema che realizzeremo è organizzato in tre moduli:

- *sistema*;
- *io*;
- *utente*.

Ogni modulo è un programma a sé stante, non collegato con gli altri due. Il modulo *sistema* contiene la realizzazione dei processi, inclusa la gestione delle memoria virtuale; il modulo *io* contiene le routine di ingresso/uscita (I/O) che permettono di utilizzare le periferiche collegate al sistema (tastiera, video, hard disk, ...). Sia il modulo sistema che il modulo *io* verranno eseguiti con il processore a livello sistema. Solo il modulo utente verrà eseguito al livello utente.

I moduli sistema e *io* forniscono un supporto al modulo utente, sotto forma di *primitive* che il modulo utente può invocare. In particolare, il modulo utente può creare più processi, che verranno eseguiti concorrentemente. I processi avranno sia una parte della memoria condivisa tra tutti, sia una parte privata per ciascuno.

1.1 Sviluppo di programmi

Siamo ormai abituati a sviluppare programmi su sistemi autosufficienti, in cui gli strumenti di sviluppo (editor, compilatore, collegatore, debugger, ...) sono a loro volta dei programmi che girano sullo stesso sistema. Quando si crea da zero un nuovo sistema, però, si passa in genere da una fase in cui si sfrutta un sistema già esistente, che chiamiamo *sistema di appoggio*. I programmi di sviluppo girano sul sistema di appoggio, ma producono eseguibili per il nuovo sistema. Tali eseguibili devono poi essere in qualche modo caricati sul nuovo sistema.

Il sistema che sviluppiamo ora non è autosufficiente e, per motivi di semplicità, non lo diventerà. Quindi per sviluppare i moduli useremo un altro sistema come appoggio. In particolare, il sistema di appoggio sarà Linux. Come compilatore utilizziamo lo stesso compilatore C++ di Linux (`g++`), opportunamente configurato in modo che produca degli eseguibili per il nostro sistema, invece che per il sistema di appoggio (come farebbe per default). In pratica, questo comporta la disattivazione di alcune opzioni, l'ordine di non utilizzare la libreria standard (in quanto userebbe quella fornita con Linux, che non funziona sul nostro sistema) e la specifica di indirizzi di collegamento opportuni. In particolare, gli indirizzi di collegamento vanno cambiati in quanto quelli di default sono pensati per i programmi utente che devono girare su Linux, rispettando quindi l'organizzazione della memoria virtuale di Linux, che è diversa da quella che utilizzeremo nel nostro sistema. Per specificare un diverso indirizzo di collegamento è sufficiente, nel nostro caso, passare al collegatore l'opzione `-Ttext` seguita da un indirizzo. Il collegatore userà quell'indirizzo come base di partenza della sezione `.text`. La sezione `.data` sarà allocata a indirizzi successivi la sezione `.text`. Per il modulo sistema useremo l'indirizzo di partenza 0x200000 (secondo MiB, per motivi spiegati in seguito).

Il modulo *io* e il modulo utente verranno gestiti tramite memoria virtuale. Per farlo è necessario suddividere in pagine le loro sezioni `.text` e `.data`, copiare tali pagine in blocchi dell'area di swap, quindi preparare tutte le tabelle di traduzione che puntino a tali blocchi e salvare anche queste tabelle nell'area di swap. In un sistema autosufficiente dotato di file system, queste operazioni (o operazioni equivalenti) vengono eseguite solo nel momento in cui si esegue un programma. Poiché, come dicevamo, il nostro sistema non è autosufficiente (e in particolare non ha un file system) creiamo l'area di swap sul sistema di appoggio. Per farlo utilizziamo il programma di utilità `creatimg`, che crea il file `swap.img`. Tale file rappresenta l'area di swap del nostro sistema, già pronta per l'avvio della memoria virtuale.

Una volta scompattato il file nucleo.tar.gz si ottiene la directory `nucleo-`$x.y$ (dove $x.y$ è il numero di versione). All'interno troviamo:

- le sottodirectory `sistema`, `io` e `utente`, che contengono i file sorgenti dei rispettivi moduli;
- la sottodirectory `util`, che contiene i sorgenti di alcuni programmi da far girare sul sistema di appoggio durante lo sviluppo dei moduli (tra cui `creatimg`);
- la sottodirectory `include`, che contiene dei file `.h` inclusi da tutti i sorgenti;
- la sottodirectory `build`, inizialmente vuota, destinata a contenere i moduli finiti;
- il file `Makefile`, contenente le istruzioni per il programma `make` del sistema di appoggio;
- uno script `run`, che permette di avviare il sistema su una macchina virtuale.

Per compilare il modulo sistema e i programi di utilità lanciare il comando `make`. Questo legge il file `Makefile` e vi trova i comandi da eseguire per costruire quanto richiesto (se lanciato senza argomenti, come in questo caso, costruisce la prima cosa menzionata nel `Makefile`).

Si noti che il programma `make` cerca di eseguire solo le operazioni strettamente necessarie. Per esempio, se lo si lancia due volte di seguito si vedrà che la prima volta verranno eseguiti tutti i diversi comandi di compilazione e collegamento, ma la seconda volta, dal momento che il modulo `sistema` esiste già e i file sorgenti non sono cambiati, non verrà eseguito alcun comando. Se si vuole forzare la ricompilazione di tutto, si può prima lanciare il comando `make clean`, che cancella tutti i file `.o` e tutto il contenuto della directory `build`. In questo modo un successivo `make` sarà costretto a rifare tutto daccapo.

Si suppone che i moduli sistema e *io* cambino raramente e costituiscano il sistema vero e proprio, mentre il modulo utente rappresenta il programma, di volta in volta di verso, che l'utente del nostro sistema vuole eseguire. Per questo motivo la sottodirectory `utente` contiene solo alcuni file di supporto (`lib.cpp`, contentente alcune funzioni di utilità, e `utente.s`, contentente la parte assembler delle chiamate di primitiva, come vedremo), e una sottodirectory `examples` contenente alcuni esempi di possibili programmi utente. La Figura 1 mostra l'esempio `mailbox.in`. L'esempio è scritto in C++ con l'aggiunta di alcune parole chiave specifiche per il nostro sistema. In particolare, alle righe 14–16 si definiscono tre processi (parola chiave `process`): il processo `scrittore1` che eseguirà la funzione `pms` con argomento `1`, il processo `scrittore2` che eseguira la stessa funzione, ma con argomento `2`, e il processo `lettore` che eseguirà la funzione `pml` con argomento 0. La funzione `pms` è definita alle righe 25–41 tramite la parola chiave `process_body`, e analogamente per la funzione `pml` alle righe 43–61. I processi possono utlizzare primitive fornite dal modulo sistema (in questo caso `sem_wait`, `sem_signal` e `delay`) e altre primitive fornite dal modulo *io* (in questo caso la sola `writeconsole`).

Per compilare questo esempio, copiarlo dalla directory `utente/examples` nella directory `utente/prog`, quindi lanciare il comando `make swap`. Si noti che questo compilerà anche il modulo `io`. Inoltre, usarà il programma `creatimg` per preparare l'area di swap, contenente le pagine estratte dai due moduli, `utente` e `io`, e le necessarie tabelle di corrispondenza.

1.2 Avvio del sistema

Una volta costruito il modulo `sistema` e il file `swap.img` possiamo avviare il sistema. All'avvio il processore parte in modalià a 16 bit non protetta (il cosiddetto "modo reale") e deve essere prima portato, via software, in modalità protetta a 32 bit. Questo compito è normalmente svolto da un programma di bootstrap caricato dal BIOS. Nel nostro caso, visto che caricheremo il sistema esclusivamente in un una macchina virtuale, questo compito sarà svolto dall'emulatore stesso. Tocca però a noi portare il processore nella modalità a 64 bit, e questo compito lo facciamo svolgere dal programma `boot` già usato per gli esempi di I/O. Si noti che la modalità a 64 bit è in realtà una sottomodalità

```
1  /*
2   * Mailbox
3   */
4
5  #include <sys.h>
6  #include <lib.h>
7
8  const int NMESG = 5;
9  const int MSG_SIZE = 100;
10
11 semaphore mailbox_piena value 0;
12 semaphore mailbox_vuota value 1;
13
14 process scrittore1 body pms(1), 5, LIV_UTENTE;
15 process scrittore2 body pms(2), 5, LIV_UTENTE;
16 process lettore   body pml(0), 5, LIV_UTENTE;
17
18 struct mess {
19         int mittente;
20         char corpo[MSG_SIZE];
21 };
22
23 mess mailbox;
24
25 process_body pms(int a)
26 {
27         char buf[MSG_SIZE];
28         char *ptr;
29         for (int i = 0; i < NMESG; i++) {
30                 ptr = copy("Messaggio numero ", buf);
31                 int_conv(i, ptr);
32                 sem_wait(mailbox_vuota);
33                 mailbox.mittente = a;
34                 copy(buf, mailbox.corpo);
35                 sem_signal(mailbox_piena);
36                 delay(20);
37         }
38         ptr = copy("fine scrittore", buf);
39         int_conv(a, ptr);
40         writeconsole(buf);
41 }
42
43 process_body pml(int a)
44 {
45         char buf[100 + MSG_SIZE], *ptr;
46         char corpo[MSG_SIZE];
47         int mittente;
48         for (int i = 0; i < 2 * NMESG; i++) {
49                 sem_wait(mailbox_piena);
50                 mittente = mailbox.mittente;
51                 copy(mailbox.corpo, corpo);
52                 sem_signal(mailbox_vuota);
53                 ptr = copy("mittente=", buf);
54                 ptr = int_conv(mittente, ptr);
55                 ptr = copy(" corpo=", ptr);
56                 copy(corpo, ptr);
57                 writeconsole(buf);
58         }
59         writeconsole("fine lettore");
60         pause();
61 }
```

Figura 1: Un esempio di programma utente.

della paginazione, quindi per portare il processore a 64 bit il programma `boot` deve abilitare la memoria virtuale. Per farlo nel modo più semplice possibile, crea le tabelle necessarie alla finestra di memoria fisica, in modo che tutta la memoria fisica sia accessibile agli stessi indirizzi a cui lo era prima di abilitare la memoria virtuale. Una volta fatto questo, il programma `boot` può cedere il controllo al modulo `sistema`. Una volta che il sistema è partito lo spazio da 0x100000 a 0x200000 può essere riutilizzato (vedremo che verrà utilizzato dallo heap di sistema). Lo spazio di memoria da 0 a 0x100000-1, invece, contiene varie cose che hanno usi specifici (per esempio, la memoria video in modalità testo). Soli i primi 640 KiB sono liberamente utilizzabili. Il programma `boot` alloca in questo spazio le tabelle di traduzione necessarie per creare la finestra di memoria fisica. Per semplicità il modulo sistema non utilizza questo spazio in altro modo.

Più in dettaglio, `boot` viene caricato dall'emulatore a partire dall'indirizzo fisico 0x100000, subito seguito da una copia del file `sistema`. Il modulo sistema è collegato a partire dall'indirizzo 0x200000 (in realtà 0x200200 per alcune limitazioni tecniche del collegatore). Il programma `boot` si preoccupa di copiare le sezioni `.text`, `.data`, etc. dalla copia del file `sistema` al loro indirizzo di collegamento, abilitare la modalità a 64 bit, quindi saltare all'entry point del modulo sistema.

In pratica, all'interno della directory del nucleo è presente lo script `run` che provvede a lanciare la macchina virtuale con tutto il necessario per far partire `boot`, quindi è sufficiente digirare `./run` per avviare il sistema.

Una volta avviato vediamo una nuova finestra che rappresenta il video della macchina virtuale. Notiamo anche dei messaggi sul terminale da cui abbiamo lanciato `./run`, qui riportati in Figura 2. Questi sono messaggi inviati sulla porta seriale della macchina virtuale. I messaggi nelle righe 1–10 arrivano dal programma `boot`. Alla riga 5 il programma `boot` ci informa del fatto che il bootloader precedente (l'emulatore stesso, nel nostro caso) ha caricato in memoria il file `build/sistema` all'indirizzo 0x109000. Nelle righe 6–9 ci riporta come sta copiando le sezioni nella loro destinazione finale. La riga 10 ci avverte che `boot` ha finito e sta per saltare all'indirizzo mostrato (0x200200), dove si trova l'entry point del modulo sistema. I messaggi successivi arrivano dal modulo sistema (alcuni, come quelli alle righe 36 e 39, arrivano dal modulo *io*). Vengono inizializzate in ordine la GDT (riga 12), lo heap di sistema (riga 13, riutilizzando lo spazio occupato da `boot`), i descrittori di pagina fisica (riga 14). Quindi il modulo sistema ci mostra come verrà suddivisa la memoria virtuale dei processi (righe 15–19) prima di caricare il registro CR3 (riga 20). Di seguito viene inizializzato l'APIC (riga 21) e lette le informazioni iniziali dall'area di swap (righe 22–26). Vengono poi creati i primi processi di sistema (righe 27–33), inizializzato il modulo *io* (righe 34–40), creato il primo processo utente (righe 41–42), attivato il timer (riga 43) e ceduto il controllo al modulo utente (riga 44). In questo esempio il modulo utente provvederà a creare i processi richiesti nel file `mailbox.in` (righe 47–49).

```
1  INF            Boot loader Calcolatori Elettronici, v0.01
2  INF            argomenti: /home/giuseppe/CE/lib/ce/boot.bin
3  INF            argv[0] = '/home/giuseppe/CE/lib/ce/boot.bin'
4  INF            mods_count = 1, mods_addr = 0x00108000
5  INF            mod[0]:build/sistema: start 0x00109000 end 0x0012c8b8
6  INF            Copiata sezione di 39000 byte all'indirizzo 00200000
7  INF            azzerati ulteriori 0 byte
8  INF            Copiata sezione di 16752 byte all'indirizzo 00409858
9  INF            azzerati ulteriori 79040 byte
10 INF            entry point 00200200
11 INF    0       Nucleo di Calcolatori Elettronici, v5.6
12 INF    0       gdt inizializzata
13 INF    0       Heap di sistema: 00100200 B @00100000
14 INF    0       Pagine fisiche: 983
15 INF    0       sis/cond [0000000000000000, 0000008000000000)
16 INF    0       sis/priv [0000008000000000, 0000010000000000)
17 INF    0       io /cond [0000010000000000, 0000018000000000)
18 INF    0       usr/cond [ffff800000000000, ffffc00000000000)
19 INF    0       usr/priv [ffffc00000000000, ffffff8000000000)
20 INF    0       Caricato CR3
21 INF    0       APIC inizializzato
22 DBG    0       lettura del superblocco dall'area di swap...
23 DBG    0       lettura della bitmap dei blocchi...
24 INF    0       sb: blocks = 20480
25 INF    0       sb: user   = _start [utente.s:104]/ffff800000201be8
26 INF    0       sb: io     = _start [io.s:60]/0000010000204a70
27 INF    0       Creato il processo dummy
28 INF    0       Creati i processi esterni generici
29 INF    0       Creato il processo main_sistema
30 INF    80      creazione o lettura delle tabelle e pagine residenti condivise...
31 INF    80      lettura del direttorio principale...
32 INF    80      creazione del processo main I/O...
33 INF    80      proc=432 entry=_start [io.s:60](0) prio=268435455 liv=0
34 INF    80      attendo inizializzazione modulo I/O...
35 INF    432     estern=448 entry=estern_kbd(int) [io.cpp:512](0) prio=1000 liv=0 type=1
36 INF    432     vid: video inizializzato
37 INF    432     estern=64 entry=estern_com(int) [io.cpp:90](0) prio=1000 liv=0 type=4
38 INF    432     estern=112 entry=estern_com(int) [io.cpp:90](1) prio=999 liv=0 type=3
39 INF    432     com: inizializzate 2 seriali
40 INF    432     estern=96 entry=esternAta(int) [io.cpp:820](0) prio=1000 liv=0 type=15
41 INF    80      creazione del processo start_utente...
42 INF    80      proc=288 entry=_start [utente.s:104](0) prio=268435455 liv=3
43 INF    80      attivato timer (DELAY=59659)
44 INF    80      passo il controllo al processo utente...
45 INF    80      Processo 80 terminato
46 INF    432     Processo 432 terminato
47 INF    288     proc=80 entry=pms(int) [mailbox.in:26](1) prio=5 liv=3
48 INF    288     proc=432 entry=pms(int) [mailbox.in:26](2) prio=5 liv=3
49 INF    288     proc=464 entry=pml(int) [mailbox.in:44](0) prio=5 liv=3
50 INF    288     Processo 288 terminato
51 INF    464     Processo 464 terminato
52 INF    80      Processo 80 terminato
53 INF    432     Processo 432 terminato
```

Figura 2: Esempio di messaggi di log inviati sulla porta seriale.