Introduzione al sistema multiprogrammato

G. Lettieri

15 Aprile 2019

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 della memoria (che, come vedremo, usa la tecnica della 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 sistema deve essere caricato dal bootstap loader, che è in grado di interpretare i file ELF e leggere il file system della macchina ospite (la macchina Linux su cui avviamo la macchina virtuale). L'output del collegatore del sistema, dunque, è direttamente utilizzabile. Il modulo *io* e il modulo utente, invece, devono essere caricati dal modulo sistema, che invece non può accedere al file system dell'ospite e in più non conosce il formato ELF. Dobbiamo dunque tradurre i moduli *utente* e *io* e trasferirli nell'hard disk della macchina virtuale. Per farlo utilizziamo il programma di utilità `creatimg`, che crea il file `swap.img`. Tale file rappresenta un hard disk della macchina virtuale (più precisamente, l'area di swap).

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.

```
1  #include <sys.h>
2  #include <lib.h>
3
4  int main()
5  {
6      writeconsole("Hello, world!");
7      pause();
8      terminate_p();
9  }
```

Figura 1: Un esempio di programma utente (file `utente/utente.cpp`).

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. In Figura 1 vediamo un esempio minimo, che può essere scritto direttamente nel file `utente/utente.cpp`. Alla riga 1 si include il file che contiene le dichiarazioni delle primitive di sistema (tra cui la dichiarazione delle primitive invocate alle righe 6 e 8). Alla riga 2 si include anche il file `lib.h`, che contiene la dichiarazione di alcune funzioni di utilità che possono essere eseguite a livello utente (ci serve in particolare la dichiarazione della funzione `pause()`. La primitiva `writeconsole()`, implementata nel modulo *io*, permette di scrivere una stringa sul monitor. Si noti la necessità di chiamare la primitiva `terminate_p()`: la funzione `main` verrà eseguita da un processo utente, che deve chiedere al sistema di poter terminare. La funzione `pause()` alla riga 7 serve solo a impedire che il sistema esegua troppo velocemente lo shutdown impedendoci di vedere la stringa stampata alla riga 6. Questo perché il sistema esegue lo shutdown non appena tutti i processi utente sono terminati.

Per compilare questo esempio lanciare il comando `make swap`, che provvederà anche a chiamare il programma `creatimg` per preparare l'area di swap

contenente i due moduli, `utente` e `io`, nel formato utilizzabile dal modulo *sistema*.

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 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 (0x200120), dove si trova l'entry

```
1  INF     0       Boot loader Calcolatori Elettronici, v0.01
2  INF     0       argomenti: /home/giuseppe/CE/lib/ce/boot.bin
3  INF     0       argv[0] = '/home/giuseppe/CE/lib/ce/boot.bin'
4  INF     0       mods_count = 1, mods_addr = 0x00108000
5  INF     0       mod[0]:build/sistema: start 0x00109000 end 0x0012d508
6  INF     0       Copiata sezione di 37104 byte all'indirizzo 00200000
7  INF     0       azzerati ulteriori 0 byte
8  INF     0       Copiata sezione di 16754 byte all'indirizzo 0020afd8
9  INF     0       azzerati ulteriori 74814 byte
10 INF     0       entry point 00200120
11 INF     0       Nucleo di Calcolatori Elettronici, v5.11
12 INF     0       gdt inizializzata
13 INF     0       Heap di sistema: 00100120 B @00100000
14 INF     0       Pagine fisiche: 1491
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:9]/ffff800000003018
26 INF     0       sb: io     = start [io.s:58]/0000010000006170
27 INF     0       Creato il processo dummy (id = 32)
28 INF     0       Creato il processo main_sistema (id = 48)
29 INF     48      creazione o lettura delle tabelle e pagine residenti condivise...
30 INF     48      lettura del direttorio principale...
31 INF     48      creazione del processo main I/O...
32 INF     48      proc=64 entry=start [io.s:58](0) prio=268435455 liv=0
33 INF     48      attendo inizializzazione modulo I/O...
34 INF     64      estern=80 entry=estern_kbd(int) [io.cpp:511](0) prio=1000 liv=0 type=1
35 INF     64      vid: video inizializzato
36 INF     64      estern=96 entry=estern_com(int) [io.cpp:89](0) prio=1000 liv=0 type=4
37 INF     64      estern=112 entry=estern_com(int) [io.cpp:89](1) prio=999 liv=0 type=3
38 INF     64      com: inizializzate 2 seriali
39 INF     64      estern=128 entry=esternAta(int) [io.cpp:822](0) prio=1000 liv=0 type=15
40 INF     64      Processo 64 terminato
41 INF     48      creazione del processo start_utente...
42 INF     48      proc=144 entry=start [utente.s:9](0) prio=268435455 liv=3
43 INF     48      attivato timer (DELAY=59659)
44 INF     48      passo il controllo al processo utente...
45 INF     48      Processo 48 terminato
46 INF     144     Processo 144 terminato
```

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

point del modulo sistema. I messaggi successivi arrivano dal modulo sistema (alcuni, come quelli alle righe 35 e 38, arrivano dal modulo *io*). Vengono inizializzate in ordine la GDT (riga 12) e lo heap di sistema (riga 13, riutilizzando lo spazio occupato da `boot`). Le righe 14–20 contengono informazioni relative alla memoria virtuale, che per il momento ignoriamo. 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 27–28). Da questo punto in poi l'inizializzazione prosegue nel processo `main_sistema` (id 48). Le righe 29 e 30 sono sempre relative alla memoria virtuale. Le righe 31–40 sono relative all'inizializzazione del modulo *io*. Viene infine creato il primo processo utente (righe 41–42), attivato il timer (riga 43) e ceduto il controllo al modulo utente (righe 44–45). In questo caso il processo utente esegue il codice di Figura 1, che stampa un messaggio sul video e poi termina (riga 46).

1.3 Uso del debugger

L'utilizzo di una macchina virtuale, e in particolare di QEMU, ci permette di collegare il debugger dalla macchina host e osservare tutto quello che accade nel sistema.

Per farlo è sufficiente avviare la macchina viruale passando l'opzione `-g` allo script `./run`. L'emulatore partirà e si sospenderà in attesa di un collegamento dal debugger. A questo punto, da un secondo terminale, si deve eseguire il comando `gdb` nella stessa directory da cui si era lanciato `./run -g`. La necessità di trovarsi nella stessa directory viene dal fatto che questa contiene uno script `.gdbinit`, che gdb legge all'avvio, con tutti i comandi necessari a collegarsi a QEMU. Si noti che la lettura automatica del file `.gdbinit` potrebbe essere stata disabilitata. Per riattivarla usare il comando:

```
echo "add auto-load-safe-path ." >> ~/.gdbinit
```

Lo script carica anche alcune estensioni scritte in python (file `debug/nucleo.py`) che mostrano automaticamente alcune informazioni e aggiungono nuovi comandi. In particolare, ogni volta che il debugger riacquisisce il controllo, viene mostrato:

- lo stack delle chiamate;
- il file sorgente nell'intorno del punto in cui si trova **`rip`**;
- se il sorgente è C++, i parametri della funzione in cui ci troviamo e tutte le sue variabili locali; altrimenti (assembler) i registri e la parte superiore della pila;
- le liste `esecuzione` e `pronti`.

Oltre ai normali comandi di gdb, sono disponibili i seguenti:

`process list`
: mostra una lista di tutti i processi attivi;

`process dump` ***id***
: mostra il contenuto (della parte superiore) della pila sistema del processo *id* e il contenuto dell'array `contesto` del suo descrittore di processo.

Altri comandi servono ad esaminare altre strutture dati che per il momento non abbiamo introdotto.