

# Accesso a basso livello

G. Lettieri

17 Marzo 2022

## 1 Ambiente protetto

Nella nostra comune esperienza di programmatori, non sembra essere vero quanto dicevamo nelle prime lezioni, e cioè che è il software a comandare mentre l'hardware, compreso il processore, si limita semplicemente ad obbedire.

In particolare, i nostri programmi per Linux non sembrano essere in grado di controllare completamente nessuna delle tre componenti fondamentali dell'hardware:

- il processore;
- l'I/O;
- la memoria;

### 1.1 Limitazioni nell'uso del processore

Per quanto riguarda il processore, i nostri programmi non sono completamente padroni del flusso di controllo. Possiamo rendercene conto se scriviamo un programma come quello in Figura 1, che esegue un ciclo infinito. Se la CPU obbedisse davvero completamente al nostro programma, il computer dovrebbe smettere di fare qualsiasi cosa nel momento in cui entra in quel ciclo. Invece, se proviamo a compilarlo ed eseguirlo, vediamo che possiamo ancora muovere il mouse, creare e spostare altre finestre, lanciare altri programmi, persino premere ctrl+C nella finestra in cui avevamo lanciato il ciclo infinito ed interromperlo.

```
int main()
{
    for (;;)
    return 0;
}
```

Figura 1: Un programma che esegue un ciclo infinito.

```

.global main
main:
    pushq %rbp
    movq %rsp, %rbp

    inb $0x60, %al

    leave
    ret

```

Figura 2: Un programma che tenta di accedere ad un indirizzo nello spazio di I/O.

## 1.2 Limitazioni nell'accesso allo spazio di I/O

Per quanto riguarda l'I/O, possiamo provare a scrivere un programma che legge da un registro di una delle periferiche. Per esempio, l'interfaccia della tastiera ha un registro RBR (Receive Buffer Register) all'indirizzo 0x60 dello spazio di I/O. Come vedremo, questo registro contiene un codice che identifica l'ultimo tasto che è stato premuto e che il software non ha ancora letto. Scriviamo il programma di Fig 2 che legge da questo registro e ne restituisce il contenuto. Il programma è scritto in Assembler, in quanto il C++ non conosce le istruzioni **in** e **out**. Il programma viene assemblato senza alcun problema, ma quando lo lanciamo viene fermato e genera un Segmentation Fault. Se ci facciamo generare il file core e lo carichiamo nel debugger, vediamo che il programma è stato fermato proprio sull'istruzione **in**. Qualcosa, dunque, impedisce al nostro programma di accedere a quel registro, e più in generale a qualunque indirizzo dello spazio di I/O.

## 1.3 Limitazioni nell'accesso alla spazio di memoria

Occupiamoci infine della memoria, ma prima dobbiamo chiarire un possibile dubbio. Qualcuno potrebbe pensare che, anche programmando in assembler, si possa solo accedere in memoria usando etichette, e dunque solo a regioni di memoria che abbiamo in qualche modo dichiarato. Non è assolutamente così: le etichette sono solo una comodità offertaci da assembler e collegatore per permetterci di usare indirizzi che o non conosciamo, perché verranno scelti dal collegatore, o che non ci interessa conoscere numericamente, perché per i nostri scopi uno vale l'altro. Ma questo non vuol dire che siamo obbligati ad usare etichette. Prendiamo, per esempio, il programma di Figura 3. Il programma dichiara una variabile `miavar` che inizialmente contiene 35 e il suo `main` ne restituisce il contenuto. Se scriviamo il programma in un file `mem.s`, lo assembliamo, lo lanciamo e poi chiediamo al sistema di mostrarci il valore restituito:

```

.data
miavar:
    .long 35

.text
.global main
main:
    pushq %rbp
    movq %rsp, %rbp

    movq miavar, %rax

    leave
    ret

```

Figura 3: Un programma che “dichiara” una variabile e ne restituisce il contenuto.

```

0000000000401106 <main>:
  401106:    55                push   %rbp
  401107:    48 89 e5          mov    %rsp,%rbp
  40110a:    48 8b 04 25 28 40 40  mov    0x404028,%rax
  401111:    00
  401112:    c9                leaveq
  401113:    c3                retq

```

Figura 4: Estratto dell’output di `objdump -d` per il programma di Figura 3.

```

g++ -o mem -no-pie mem.s
./mem
echo $?

```

otteniamo effettivamente 35. Fin qui niente di nuovo. Ora usiamo il programma `nm` per ottenere l’indirizzo che il collegatore ha assegnato a `miavar`:

```
nm mem | grep miavar
```

Nel mio caso l’indirizzo è `0x404028`. In Figura 4 possiamo vedere il disassemblato del programma finale e confermare che l’istruzione `movq miavar, %rax` contiene effettivamente `0x404028` nel suo campo indirizzo (si ricordi che l’architettura è little-endian). Ora modifichiamo il programma in modo da fargli usare direttamente l’indirizzo `0x404028` e non l’etichetta, ottenendo la versione di Figura 5. Se assembliamo, lanciamo e mostriamo il contenuto di `?` otteniamo 35 esattamente come prima. Anche l’esame del disassemblato mostrerà che l’eseguibile ottenuto dal programma di Figura 5 è esattamente identico a quello ottenuto da Figura 3: Le etichette, dunque, servono ad evitare di dover scrivere esplicitamente gli indirizzi, ma il modello di programmazione del linguaggio

```

.data
    .long 32
.text
.global main
main:
    pushq %rbp
    movq %rsp, %rbp

    movq 0x404028, %rax

    leave
    ret

```

Figura 5: Lo stesso programma di Figura 3, ma senza l’etichetta `miavar`.

```

.data
    .long 32
.text
.global main
main:
    pushq %rbp
    movq %rsp, %rbp

    movq 0x401106, %rax

    leave
    ret

```

Figura 6: Lo stesso programma di Figura 5, ma con un diverso indirizzo.

macchina (e dunque dell’assembler) non prevede che il programmatore “dichiari” variabili, ma che acceda liberamente agli indirizzi della memoria. Possiamo anche modificare il programma come in Figura 6 per leggere il byte che si trova all’indirizzo `0x401106` (il primo byte di Figura 4). Questa volta `?` conterrà `85` (`0x55` in decimale).

Se si prova a variare l’indirizzo si scoprirà che Linux permette al nostro programma di accedere liberamente ad alcuni indirizzi, ma non a tutti. Possiamo vedere quali indirizzi sono stati assegnati al nostro programma caricandolo nel debugger, inserendo un breakpoint su `main`, avviandolo e infine usando il comando `info proc mappings`. Un qualunque indirizzo al di fuori degli intervalli `[StartAddr, EndAddr)` causerà un `Segmentation Fault`. Inoltre, ad alcuni intervalli si ha accesso solo in lettura. In particolare, se proviamo a modificare il programma di Figura 6 in modo che *scriva* all’indirizzo `0x401106` invece di leggervi, causeremo un `Segmentation Fault` anche in questo caso. Questo perché

Linux vieta le scritture nella sezione `.text`.

## 2 Ambiente non protetto

Le precedenti limitazioni sono imposte dal kernel (nucleo) di Linux utilizzando tre meccanismi:

- interruzioni;
- protezione;
- memoria virtuale.

Questi sono meccanismi hardware che si aggiungono a quelli che abbiamo visto fino ad ora. Il kernel Linux è un software che usa questi meccanismi per permettere a più programmi di essere eseguiti contemporaneamente (almeno in apparenza) e in modo sicuro. Il kernel Linux stesso non è soggetto a nessuna delle limitazioni che abbiamo osservato nella sezione 1. Ciò è reso possibile dal fatto che, all'avvio, l'hardware è configurato per non imporre alcuna restrizione al software. Il primo programma che viene caricato, dunque, è effettivamente il padrone dell'hardware, così come lo abbiamo immaginato nelle prime lezioni. Questo software (per es., il kernel Linux, o quello di Windows o di Mac OS) sfrutta il proprio potere per riconfigurare l'hardware, usando i tre meccanismi di cui sopra, in modo che il resto del software (i nostri programmi) sia soggetto a tutte le limitazioni della sezione 1.

Se vogliamo studiare questi meccanismi e osservare cosa l'hardware permette realmente di fare al software, dobbiamo scrivere programmi che vengano caricati direttamente all'avvio del sistema, *al posto* del kernel del nostro sistema operativo. Farlo su un computer reale, però, è tecnicamente molto complesso, oltre che scomodo e pericoloso. Per farlo in pratica conviene utilizzare una *macchina virtuale* che emuli in tutto e per tutto una sistema reale con la sua CPU, la sua memoria e i suoi dispositivi di I/O, ma nella quale sia molto più semplice caricare un programma a nostra scelta.

Useremo l'emulatore QEMU opportunamente configurato, in modo che ci fornisca il seguente hardware (emulato):

- un processore Intel/AMD a 64 bit, come quello che stiamo studiando;
- una memoria RAM di dimensione configurabile (16 MiB per default);
- una tastiera, emulata tramite la tastiera del computer stesso;
- un monitor, emulato tramite una finestra del nostro sistema operativo;
- un hard disk, emulato usando un file del nostro sistema operativo (per default il file `CE/share/hd.img` nella directory home);
- varie interfacce I/O che vedremo in seguito, compatibili con quelle di un comune PC.

L'emulatore QEMU è anche in grado di caricare nella RAM della macchina virtuale, all'avvio, un file eseguibile letto dal nostro sistema operativo (scavalcando l'emulazione della ROM di bootstrap nella macchina virtuale). Questo è estremamente comodo per noi, perché vuol dire che possiamo creare questo file lavorando sul nostro sistema reale. Come se non bastasse, il file che QEMU è in grado di caricare può essere in formato ELF, quindi possiamo crearlo usando lo stesso compilatore, assembler e collegatore di cui già disponiamo e che abbiamo usato fino ad ora. Dobbiamo soltanto stare attenti al fatto che questi strumenti, così come li troviamo installati sul nostro sistema Linux, sono stati configurati per creare file eseguibili dal kernel Linux. In particolare, il collegatore assegnerà indirizzi legati alla memoria virtuale di Linux, e collegherà automaticamente la libreria del C++, la quale userà le funzionalità offerte dal kernel Linux per scrivere sullo schermo, leggere dalla tastiera e mille altre cose. Quando il nostro programma verrà caricato da QEMU, però, sarà lui il "kernel" e dentro la macchina virtuale non ci sarà altro software che il nostro, e meno che mai ci sarà il kernel Linux. Dobbiamo dunque passare diverse opzioni ai vari strumenti in modo che non facciano le cose che non hanno alcun senso nel nostro caso, e in particolare che non colleghino la libreria del C++<sup>1</sup>

In assenza di qualunque libreria, però, è estremamente oneroso scrivere un programma che inizializzi tutto ciò che va inizializzato e permetta anche di interagire con l'utente, anche solo tramite tastiera e video. Per questo motivo useremo una nostra libreria, `libce`, che esegue queste inizializzazioni prima di chiamare `main` e fornisce alcune funzioni già pronte per l'I/O. Inoltre, quando `main` ritorna, la libreria si preoccupa di spegnere la macchina virtuale.

Lo script `compile` passa tutte le opzioni necessarie ai vari strumenti e collega il programma con `libce`. Un altro script, `boot`, avvia l'emulatore configurato come abbiamo detto sopra, dicendogli di caricare in memoria il nostro programma. Da quel punto in poi il nostro programma è finalmente il vero padrone, anche se solo all'interno della macchina virtuale. Il fatto che la macchina sia virtuale ci fornisce un'altra opportunità molto comoda, che sarebbe molto difficile se non impossibile avere su un sistema reale: collegare un debugger al sistema e osservare cosa sta facendo la CPU della macchina virtuale. Per far questo è necessario passare l'opzione `-g` allo script `boot`, quindi usare (in un altro terminale) lo script `debug`, che invoca `gdb` passandogli tutte le informazioni necessarie per collegarsi alla macchina virtuale. Per usare questi script (`compile`, `boot` e `debug`) conviene creare una directory che contenga solo i sorgenti del programma che vogliamo caricare, che possono essere sia in C++ che in assembler, quindi lanciare gli script da quella directory.

---

<sup>1</sup>Si noti che questo ci impedirà di usare anche alcune funzionalità del C++ che hanno bisogno di questa libreria, come `new` e `delete`, le eccezioni e la Run Time Type Information (RTTI) usata, tra l'altro, dalle eccezioni e dal `dynamic_cast`. Alcune di queste funzionalità, come `new` e `delete`, possono essere re-implementate con poco sforzo, mentre le altre sono improponibili.

```

#include <libce.h>
int main()
{
    natb c;
    inputb(0x60, c);
    printf("%2x\n", c);
    pause();
    return 0;
}

```

Figura 7: Un programma che legge un byte dallo spazio di I/O.

## A Esempi di programmi in ambiente non protetto

Vediamo ora come, usando l'ambiente non protetto, tutte le limitazioni della sezione 1 sono disabilitate.

### A.1 Completo controllo della CPU

In una directory `cpu` scriviamo lo stesso programma di Figura 1, quindi entriamo nella directory (`cd cpu`) e lanciamo `compile` e `boot -g`. La macchina virtuale parte e si mette in attesa del collegamento da `gdb`. Da un altro terminale, portiamoci nuovamente nella directory `cpu` e lanciamo lo script `debug`. Facciamo proseguire l'emulazione (comando `c`) fino all'inizio del `main`, quindi seguiamo eseguendo una singola istruzione di linguaggio macchina alla volta (comando `si`). Vediamo come la CPU emulata continua ad eseguire le istruzioni del ciclo infinito indefinitamente. Si noti che l'emulatore ci permette di osservare *tutto* ciò che la CPU emulata sta facendo. Possiamo premere `Ctrl+C` nella finestra in cui avevamo lanciato `boot -g` per interrompere l'emulatore (anche questo è un vantaggio dell'usare un emulatore: nella macchina reale avremmo dovuto riavviare).

### A.2 Completo controllo dell'I/O

Vedremo meglio in seguito che possiamo programmare tutte le periferiche della macchina virtuale a nostro piacimento. Per il momento proviamo ad eseguire l'analogo del programma di Figura 2, ma stampando sul video il valore letto dal registro RBR della tastiera, invece che restituendolo (all'interno della macchina virtuale non c'è nessun sistema operativo a cui restituirlo). Per leggere il registro usiamo la funzione `inputb()`, che è scritta in assembler e usa l'istruzione `inb`. Per stampare qualcosa sul video usiamo la funzione `printf()` fornita da `libce` e analoga alla funzione omonima della libreria del C, alla cui

```

#include <libce.h>
int main()
{
    natb *mem = (natb*)4096;
    unsigned long i;
    for (i = 0; i < 16*1024*1024; i += 1024)
        printf("%2x ", mem[i]);
    pause();
    return 0;
}

```

Figura 8: Un programma che legge e stampa byte presi da tutta la memoria.

documentazione rimandiamo. Usiamo anche la funzione `pause()`, che attende che venga premuto il tasto ESC, per evitare che la macchina virtuale si spenga troppo velocemente non permettendoci di osservare l'output. Il programma completo è in Figura 7. Tutte le funzioni che abbiamo usato sono definite nella `libce`, e per questo includiamo il suo file di intestazione (prima riga). Anche il tipo `natb` è definito nella libreria e corrisponde ad **unsigned char**. Se proviamo a compilare e lanciare il programma nella macchina virtuale vediamo che il programma esegue correttamente, stampando due cifre esadecimali. Le cifre corrispondono in effetti al byte letto da RBR, anche se per il momento dobbiamo crederci sulla fiducia. La cosa importante è che niente ha fermato il programma mentre tentava di leggere.

### A.3 Completo controllo della memoria

Il programma di Figura 8 legge tutta la memoria RAM<sup>2</sup> della macchina virtuale, a intervalli di 1 KiB, e la mostra sul video in esadecimale. Anche in questo caso possiamo provare a lanciare il programma nella macchina virtuale e osservare come riesca ad arrivare fino in fondo senza che niente lo blocchi.

Possiamo fare anche di più: scrivere nella parte di memoria che contiene il nostro stesso programma e modificarlo. Si veda, per esempio, il programma di Figura 9 e si cerchi di capire cosa sta facendo. La cosa importante da osservare, in ogni caso, è che le linee 16 e 17 stanno scrivendo nella zona di memoria dove si trova la funzione `foo()` e, anche questa volta, niente glielo impedisce.

---

<sup>2</sup>Per motivi che vedremo in seguito evitiamo di leggere i primi 4 KiB.

```

1 #include <libce.h>
2
3 void foo(long a, long b)
4 {
5     printf("a = %d, b = %d\n", a, b);
6 }
7
8 int main()
9 {
10     foo(10, 20);
11
12     natb *b1 = (natb*)foo + 11,
13           *b2 = (natb*)foo + 15,
14           tmp;
15     tmp = *b1;
16     *b1 = *b2;
17     *b2 = tmp;
18
19     foo(10, 20);
20
21     pause();
22     return 0;
23 }

```

Figura 9: Un programma che modifica se stesso.