Introduzione al sistema multiprogrammato

G. Lettieri

1 Aprile 2022

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. Definiamo prima tre termini che ricorrono spesso in questo ambito, e di cui anche noi faremo grande uso: processo, primitiva e contesto.

1.1 I processi

L'astrazione più importante introdotta dal sistema è quella dei *processi*. Un processo è un programma in esecuzione.

Proviamo ad illustrare la differenza tra programma e processo con una semplice metafora: in una pizzeria, il pizzaiolo riceve una ordinazione. Il pizzaiolo è inesperto e ha bisogno di avere la ricetta della pizza davanti a se. Stende la pasta, la condisce, la inforna, aspetta che sia cotta, la farcisce e serve la pizza.

Il programma è la ricetta. Il pizzaiolo è il processore, cioè l'entità che interpreta la ricetta e la esegue. La pizza sono i dati da elaborare.

Il processo è un concetto un po' più astratto. È la *sequenza di stati* che il sistema "pizza + pizzaiolo" attraversa, passando dalla pasta alla pizza finale, secondo le istruzioni dettate dalla ricetta, eseguite pedissequamente dal pizzaiolo inesperto. Possiamo immaginarlo come il "filmato" del pizzaiolo che fa la pizza.

Uscendo dalla metafora, un processo è un programma in esecuzione su dei dati di ingresso. Questa esecuzione la possiamo modellare come la sequenza degli stati attraverso cui il sistema processore + memoria passa eseguendo il programma, su quei dai, dall'inizio fino alla conclusione. L'esecuzione di ogni istruzione del programma fa passare il processo da uno stato al successivo.

Notate che questa definizione si applica bene ai programmi di tipo *batch*, in cui gli ingressi vengono specificati tutti all'inizio, e il processo (generato dal programma in esecuzione su quei dati) prosegue indisturbato fino ad ottenere le uscite (ad esempio, pensate ad un programma per ordinare alfabeticamente un file). Una volta afferrato il concetto, però, in questo caso semplice, credo che non vi sarà difficile estenderlo ai programmi "interattivi" a cui ormai siamo più abituati (praticamente tutti i programmi con interfaccia grafica, ma non solo quelli).

A prima vista, il processo potrebbe sembrare molto simile al programma: la ricetta dice "stendere la pasta" e nel processo vediamo il pizzaiolo stendere la pasta; nel rigo successivo la ricetta ricetta dice "versare il condimento" e nel processo vediamo, subito dopo, il pizzaiolo versare il condimento. È molto semplice confondere i due concetti, soprattutto se il programma è molto semplice, ma si tratta di due cose completamente distinte, per i motivi illustrati di seguito.

- Uno stesso programma può essere associato a più processi: tanti clienti, in genere, chiedono lo stesso tipo di pizza. In questo caso, il programma è sempre lo stesso (la ricetta per quel tipo di pizza), ma ad ogni pizza corrisponde un processo distinto, che si svolge autonomamente nel tempo.

- Uno stesso processo può eseguire, in sequenza, più programmi. Si pensi, per esempio, ad una pizza *a metro*, composta da vari tipi di pizza uno dietro l'altro[1].

- In generale, non è esclusivamente il programma (la ricetta) a decidere attraverso quali stati il processo dovrà passare, ma anche la richiesta del cliente (l'input): la ricetta potrebbe, infatti, prevedere delle varianti (un `if`), che il cliente dovrà specificare. La ricetta conterrà istruzioni per entrambe le varianti (con o senza carciofi), ma un particolare processo seguirà necessariamente *una sola* variante.

- Il programma potrebbe contenere dei cicli (aggiungere pomodoro fino a coprire la parte centrale della pasta); nel programma vediamo le azioni da ripetere scritte una volta sola, mentre nel processo vediamo le azioni effettivamente ripetute tante volte.

Ma c'è dell'altro, nella metafora del pizzaiolo, che ci può aiutare a capire altri punti fondamentali. Il processo, se lo guardiamo nella sua interezza, si svolge necessariamente nel tempo ("procede" nel tempo). Possiamo anche, però, guardarlo ad un certo istante, facendone una fotografia, o osservando un singolo fotogramma del filmato di cui sopra. La fotografia che ne facciamo deve contenere tutte le informazioni necessarie a capire come il processo si svolgerà nel seguito. Nel nostro esempio, la foto dovrà contenere:

- la pizza nel suo stato semilavorato (la memoria dati del processo);

- il punto, sulla ricetta, a cui il pizzaiolo è arrivato (il contatore di programma);

- tutte le altre informazioni che permettono al pizzaiolo di proseguire (da quel punto in poi) con la corretta esecuzione della ricetta, come, ad esempio, il tempo trascorso da quando ha infornato la pizza (i registri del processore).

[1] Nel sistema Unix, per esempio, uno processo può interrompere il programma che sta eseguendo e passare ad un altro, invocando la primitiva `execve()`.

Se la fotografia è fatta bene, contiene tutto il necessario per sospendere il processo e riprenderlo in un secondo tempo. Il pizzaiolo fa questa operazione continuamente, quando condisce, a turno, più pizze, o quando lascia una serie di pizze nel forno e, nel frattempo, comincia a prepararne un'altra (multiprogrammazione).

I processi sono rappresentati all'interno del sistema tramite una serie di strutture dati, la più importante delle qualli è il *descrittore di processo*, una struttura che viene instanziata per ogni nuovo processo e che contiene, in particolare, lo spazio per memorizzare una istantanea dello stato del processo. Quando il sistema decide di portare avanti un processo P_1, per prima cosa carica questa istantanea nei veri registri del processore e nella vera memoria del sistema. A questo punto la normale esecuzione delle istruzioni farà avanzare lo stato del processo P_1. Quando il sistema decide di passare ad un altro processo P_2, per prima cosa scatta una nuova istantanea dello stato di P_1, che andrà a sostituire la precedente, e poi caricherà l'istantanea dello stato del processo P_2.

1.2 Primitive

L'aggettivo "primitivo/a", usato anche come sostantivo (normalmente al femminile), ricorre spesso in informatica. Con questo termine non ci si riferisce a qualcosa di "antico" o non evoluto, ma a qualcosa di *non derivato*, non ulteriormente scomponibile, fornito dal sistema come "mattone" fondamentale per costruire il resto delle cose. Per esempio, in un programma di disegno, sono chiamate "primitive" le funzioni di base fornite dal programma, come tracciare una linea, un quadrato, un cerchio. Tutto il disegno deve essere costruito usando queste primitive. Quando diciamo "non ulteriormente scomponibili" intendiamo dire che la primitiva va presa nella sua interezza, o tutta o niente. Per esempio, se la primitiva "disegna un quadrato" mi permette solo di specificare due angoli opposti e poi fa apparire tutto il quadrato, non posso usarla (da sola) per disegnare solo due lati. Un altro esempio sono i "tipi primitivi" di un linguaggio, come **`int`**, **`long`**, **`char`**, etc. in C e C++. Il programmatore può usarli per definire nuovi tipi derivati (strutture, classi, array, ...), ma non può modificare il comportamento dei tipi primitivi stessi.

Inoltre, in genere il semplice utente di un sistema non può (o, comunque, si suppone che non debba) aggiungere altre primitive. Per esempio, un semplice utente programmatore di un linguaggio non può definire nuovi tipi primitivi (anche se, in C++, può definire classi che si comportano in modo molto simile), e solo chi definisce il linguaggio o scrive il compilatore può farlo.

Questo stesso concetto si applica anche ai sistemi multiprogrammati. Vogliamo, infatti, che questi sistemi forniscano delle operazioni con cui gli utenti possano creare e terminare nuovi processi e farli interagire tra loro, oppure eseguire operazioni di ingresso/uscita. Non vogliamo, però, che gli utenti possano interferire con il comportamento di queste operazioni, né vogliamo che ne possano definire di nuove oltre quelle pensate dai progettisti del sistema (perché le nuove operazioni potrebbero aggirare i vincoli imposti dalle altre).

Si noti che, per forza di cose, le operazioni di creazione e terminazione di un processo dovranno accedere alle strutture dati associate ai processi, come i

descrittori di processo di cui abbiamo parlato sopra. Dalla corretta gestione di queste strutture dati dipende però tutta la multiprogrammazione del sistema, dunque i programmi utente non devono potervi accedere liberamente. In qualche modo, dunque, gli utenti devono poter causare accessi a queste strutture dati, ma solo in modo controllato. In particolare, vi devono poter accedere soltanto tramite una primitiva.

1.3 Contesto

L'idea unificante, che permette di realizzare sia i processi, sia le primitive, è quella di *contesto*, a cui abbiamo già accennato. Con questa parola ci si riferisce, nel linguaggio comune come in quello tecnico che ci riguarda, a tutto ciò che è necessario sapere per interpretatare correttamente un dato testo, ma che non è scritto esplicitamente nel testo stesso.

Nel nostro caso, quando diciamo "testo" stiamo pensando al testo di un programma da eseguire. In un sistema multiprocesso il significato di una istruzione dipendende dal processo che la sta eseguendo. Per esempio, se un processo P_1 esegue una istruzione

```
mov %rax, 1000
```

si sta riferendo al "suo" registro `%rax`, il cui aveva presumibilmente scritto qualcosa in un passo precedente, e sta tentando di copiarla al "suo" indirizzo 1000, dove avrà presumibilmente allocato una qualche sua variabile. La stessa identica istruzione, eseguita però da un altro processo P_2, parlerà di un diverso `%rax` e di una diversa variabile. La corretta interpretazione dell'istruzione dipende dunque da qualcosa che non è scritto nell'istruzione: il processo che la esegue. Possiamo dunque pensare che ogni processo abbia un suo contesto. Il sistema deve essere organizzato in modo tale che, ogni volta che si esegue una qualunque istruzione, si tenga correttamente conto del contesto del processo a cui quella istruzione appartiene. Il contesto di un processo comprenderà, sicuramente, tutta la memoria usata dal processo e una copia privata di tutti i registri del processore. L'operazione di caricamento dello stato di un processo (l'istantanea di cui abbiamo parlato nella sezione precedente) non fa altro che rendere *corrente*, o attivo, il contesto di quel processo. Da quel momento in poi, e fino al prossimo cambio di contesto, le istruzioni eseguite dal processore opereranno implicitamente nel contesto di quel processo.

L'idea di contesto ci aiuta anche a capire come realizzare le primitive. Prendiamo, per esempio, una istruzione che scriva qualcosa in un descrittore di processo, e chiediamoci se è lecita oppure no, cioè se il processore deve eseguirla oppure rifiutarsi e sollevare invece una eccezione. Notiamo subito che, di per sé, l'istruzione non è né lecita né illecita: la sua liceità dipende dal contesto. Sarà lecita se ad eseguirla è il codice del sistema (quello scritto dagli operatori, nel nostro esempio originario) e illecita se la troviamo nel codice scritto dagli utenti. Questa idea si traduce nel creare un contesto *privilegiato* e uno o più non privilegiati, e facendo in modo che quella istruzione sia lecita solo nel contesto

privilegiato. Le primitive che il sistema mette a disposizione degli utenti saranno eseguite nel contesto privilegiato, mentre i programmi degli utenti saranno eseguiti in uno dei contesti non privilegiati.

Sfruttando il meccanismo della protezione, possiamo far coincidere il contesto privilegiato con la modalità sistema del processore, e i contesti non privilegiati con la modalità utente. Il sistema sarà dunque organizzato nel seguente modo:

- i programmi utente vengono eseguiti con il processore a livello utente;
- le stutture dati critiche (per es., la IDT e i descrittori di processo) vengono rese inaccessibili da livello utente (devono essere allocate in una parte della memoria a cui il processore non possa accedere da livello utente);
- il programmatore di sistema scrive delle funzioni che svolgono le operazioni per conto dell'utente (per es., creare un processo), assicurandosi di manipolare correttamente le strutture dati critiche;
- il programmatore di sistema permette agli utenti di invocare le sue funzioni esclusivamente tramite *gate* della IDT (dunque tramite l'istruzione **`int`**) che innalzino il livello del processore (portandolo a sistema).

Mentre sono in esecuzione le funzioni scritte dal programmatore di sistema, e solo allora, il processore si trova a livello sistema e può manipolare le strutture dati critiche, altrimenti queste sono inaccessibili.

2 Un semplice sistema multiprocesso

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, in un contesto privilegiato. 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.

2.1 Sviluppo di programmi

Il sistema che sviluppiamo 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). Come abbiamo già visto per gli esempi di I/O, 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. 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 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 agli indirizzi che seguono 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 QEMU). L'output del collegatore del sistema, dunque, è direttamente utilizzabile. Il boot loader carica in memoria anche il modulo *io* e il modulo utente, ma si limita a copiarli senza interpretarne il contenuto. Sarà il modulo sistema, durante la fase di inizializzazione, a interpretare i due file in modo che le varie sezioni al loro interno si trovino agli indirizzi corretti.

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;
- la sottodirectory `include`, che contiene dei file `.h` inclusi dai vari 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 <all.h>
2
3  int main()
4  {
5      writeconsole("Hello, world!\n", 14);
6      pause();
7      terminate_p();
8  }
```

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

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 diverso, che l'utente del nostro sistema vuole eseguire. Per questo motivo la sottodirectory `utente` contiene solo alcuni file di supporto (`lib.cpp` e `lib.h`, contententi 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 un file che contiene le dichiarazioni delle funzioni di libreria e delle primitive di sistema (tra cui la dichiarazione delle primitive invocate alle righe 5 e 8). Il file, in realtà, si limita ad includere vari altri file, tra cui quelli contenuti nella directory `include`, il file `lib.h` e il file di intestazione di libce. La funzione `pause()`, invocata alla linea 6, è implementata nel file `lib.cpp`. La primitiva `writeconsole()`, implementata nel modulo *io* e dichiarata in `include/io.h`, 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 6 serve solo a impedire che il sistema esegua troppo velocemente lo shutdown impedendoci di vedere la stringa stampata alla riga 5. Questo perché il sistema esegue lo shutdown non appena tutti i processi utente sono terminati.

Per compilare i moduli e i programi di utilità lanciare il comando `compile`, già usato per gli esempi di I/O[2]

[2]Nel caso del nucleo, lo script `compile` usa `make`, che può anche essere usato direttamente. Il comando `make` legge a sua volta il file `Makefile` e vi trova i comandi da eseguire per costruire quanto richiesto. 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 i moduli esistono 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 reset`, 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. Lo script `compile` esegue un `make reset` seguito da `make`.

2.2 Avvio del sistema

Una volta costruiti tutti i moduli, possiamo avviare il sistema. La procedura di *boostrap* è la stessa già usata per gli esempi di I/O e può essere avviata lanciando lo script `boot`.

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.bin` fornito da `libce`[3] Una volta fatto questo, il programma `boot.bin` può cedere il controllo al modulo `sistema`. Lo spazio da 0x100000 a 0x200000 può essere ora 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. Per semplicità il modulo sistema non utilizza questo spazio in alcun modo.

Più in dettaglio, `boot.bin` viene caricato da QEMU a partire dall'indirizzo fisico 0x100000, subito seguito da una copia dei file `sistema`, `io` e `utente`. Il modulo sistema è collegato a partire dall'indirizzo 0x200000. Il programma `boot.bin` 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.

Una volta avviato vediamo una nuova finestra che rappresenta il video della macchina virtuale. Notiamo anche dei messaggi sul terminale da cui abbiamo lanciato `boot`, qui riportati in Figura 2. Questi sono messaggi inviati sulla porta seriale della macchina virtuale. I messaggi nelle righe 1–9 arrivano dal programma `boot.bin`. Alla riga 5 il programma `boot.bin` ci informa del fatto che il bootloader precedente (QEMU stesso, nel nostro caso) ha caricato in memoria il file `build/sistema` all'indirizzo 0x10a000. Nelle righe 6–8 ci informa su come sta copiando le sezioni nella loro destinazione finale. La riga 9 ci avverte che `boot.bin` ha finito e sta per saltare all'indirizzo mostrato (0x200120), dove si trova l'entry point del modulo sistema. I messaggi successivi arrivano dal modulo sistema (alcuni, come quelli alle righe 42–44, arrivano dal modulo *io*). Vengono inizializzate in ordine la GDT (riga 11) e l'APIC (riga 12). Le righe 13–33 contengono informazioni relative alla memoria virtuale, che per il momento ignoriamo. Di seguito viene inizializzato lo heap di sistema (riga 34, riutilizzando lo spazio occupato da `boot.bin`). Vengono poi creati i primi processi di sistema (righe 35, 36 e 39). Da questo punto in poi l'inizializzazione prosegue nel processo `main_sistema` (id 0) e `main I/O` (id 2). Le righe 41–47 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

[3] I sorgenti sono in `boot64/boot.S` e `boot64/boot.cpp`.

```
1  INF     -       Boot loader Calcolatori Elettronici, v0.02
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 = 3, mods_addr = 0x00109000
5  INF     -       mod[0]:build/sistema: start 0x0010a000 end 0x0013c310
6  INF     -       Copiato segmento di 47096 byte all'indirizzo 00200000
7  INF     -       Copiato segmento di 524 byte all'indirizzo 0020cfe0
8  INF     -       ... azzerati ulteriori 78492 byte
9  INF     -       entry point 00200120
10 INF     -       Nucleo di Calcolatori Elettronici, v6.6
11 INF     -       GDT inizializzata
12 INF     -       APIC inizializzato
13 INF     -       Numero di frame: 545 (M1) 7647 (M2)
14 INF     -       sis/cond [0000000000000000, 0000008000000000)
15 INF     -       sis/priv [0000008000000000, 0000010000000000)
16 INF     -       io /cond [0000010000000000, 0000018000000000)
17 INF     -       usr/cond [ffff800000000000, ffffc00000000000)
18 INF     -       usr/priv [ffffc00000000000, 0000000000000000)
19 INF     -       Creata finestra sulla memoria centrale:  [0000000000001000, 0000000002000000)
20 INF     -       Creata finestra per memory-mapped-IO:    [00000000fec00000, 0000000100000000)
21 INF     -       mappo il modulo I/O:
22 INF     -        - segmento sistema read-only  mappato a [0000010000000000, 0000010000004000)
23 INF     -        - segmento sistema read/write mappato a [0000010000010000, 0000010000021000)
24 INF     -        - heap:                                 [0000010000021000, 0000010000121000)
25 INF     -        - entry point: start [io.s:8]
26 INF     -       mappo il modulo utente:
27 INF     -        - segmento utente  read-only  mappato a [ffff800000000000, ffff800000002000)
28 INF     -        - segmento utente  read/write mappato a [ffff800000002000, ffff800000004000)
29 INF     -        - heap:                                 [ffff800000004000, ffff800000104000)
30 INF     -        - entry point: start [utente.s:10]
31 INF     -       Create le traduzioni per le parti condivise
32 INF     -       Frame liberi: 7098 (M2)
33 INF     -       CR3 caricato
34 INF     -       Heap di sistema: 00100000 B @00100000
35 INF     -       Creato il processo main_sistema (id = 0)
36 INF     -       Creato il processo dummy (id = 1)
37 INF     0       Timer attivato (DELAY=59659)
38 INF     0       proc=2 entry=start [io.s:8](1024) prio=-1 liv=0
39 INF     0       Creato il processo main I/O (id = 2)
40 INF     0       attendo inizializzazione modulo I/O...
41 INF     2       estern=3 entry=estern_kbd(int) [io.cpp:127](0) prio=1104 (tipo=50) liv=0 irq=1
42 INF     2       kbd: tastiera inizializzata
43 INF     2       vid: video inizializzato
44 INF     2       bm: 00:01:01
45 INF     2       estern=4 entry=esternAta(int) [io.cpp:359](0) prio=1120 (tipo=60) liv=0 irq=14
46 INF     2       Processo 2 terminato
47 INF     0       ... inizializzazione modulo I/O terminata
48 INF     0       proc=5 entry=start [utente.s:10](0) prio=-1 liv=3
49 INF     0       Creato il processo start_utente (id = 5)
50 INF     0       passo il controllo al processo utente...
51 INF     0       Processo 0 terminato
52 INF     5       Processo 5 terminato
```

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

```
1  #include <all.h>
2
3  int main()
4  {
5      volatile natw* video = reinterpret_cast<natw*>(0xb8000);
6      video[4] = 0x3F00 | 'a';
7      pause();
8      terminate_p();
9  }
```

Figura 3: Un esempio di programma utente che tenta di esegure un'azione illecita.

```
1  INF     0       proc=5 entry=start [utente.s:10](0) prio=-1 liv=3
2  INF     0       Creato il processo start_utente (id = 5)
3  INF     0       passo il controllo al processo utente...
4  INF     0       Processo 0 terminato
5  WRN     5       Eccezione 14 (page fault), errore 00000007, rip main [utente.cpp:6]
6  WRN     5         indirizzo virtuale: 00000000000b8008
7  WRN     5         dettagli: protezione, scrittura, da utente,
8  WRN     5       proc 5, livello UTENTE, precedenza -1
9  WRN     5         RIP=main [utente.cpp:6] CPL=LIV_UTENTE
10 WRN     5         RFLAGS=0000000000000202 [-- -- -- IF -- -- -- -- -- --, IOPL=SISTEMA]
11 WRN     5         RAX=00000000000b8008 RBX=ffff800000002fe0 RCX=0000000000000000 RDX=ffff800000004000
12 WRN     5         RDI=ffff800000004000 RSI=0000000000100000 RBP=fffffffffffffff0 RSP=fffffffffffffe0
13 WRN     5         R8 =0000000000000000 R9 =0000000000000000 R10=0000000000000000 R11=0000000000000000
14 WRN     5         R12=ffff800000002fe0 R13=0000000000000000 R14=0000000000000000 R15=0000000000000000
15 WRN     5         backtrace:
16 WRN     5       Processo 5 abortito
```

Figura 4: Esempio di messaggi di log relativi alla terminazione forzata di un processo in seguito al sollevamento di una eccezione.

modulo utente (righe 48–50). In questo caso il processo utente esegue il codice di Figura 1, che stampa un messaggio sul video e poi termina (riga 52).

In Figura 3 mostriamo un altro esempio di programma utente, che questa volta tenta di eseguire un'azione non permessa: scrivere direttamente sulla memoria video (linea 6) senza invocare la primitiva `writeconsole()`. Il tentativo causa il sollevamento di una eccezione che restitiuisce il controllo al modulo sistema, il quale termina forzatamente il processo e invia alcuni messaggi sul log (Figura 4). I messaggi alle righe 5–15 contengono informazioni sull'errore intercettato e sullo stato del processo al momento dell'errore. In particolare, alla fine della riga 5 e all'inizio della riga 9, ci mostra il contenuto di RIP, già ricondotto alla corrispondente riga del file sorgente (in questo caso è la riga 6 del file `utente.cpp`). Le righe 10–14 mostrano anche il contenuto di tutti gli altri registri, mentre a partire dalla riga 15 viene mostrato il cosiddetto "backtrace", ovvero la pila delle chiamate di funzione ancora attive al momento dell'errore (in questo caso, dal momento che l'errore era proprio in `main()`, non ci sono altre funzioni sullo stack).

2.3 Uso del debugger

Anche in questo caso, come per gli esempi di I/O, possiamo sfruttare la possibilità di collegare il debugger dalla macchina host e osservare tutto quello che

accade nel sistema.

La procedura è quella già vista: avviamo la macchina virtuale passando l'opzione `-g` allo script `boot`; quindi, da un altro terminale, ci portiamo nella stessa directory e lanciamo lo script `debug`. Lo script, oltre alle estensioni già viste, carica altre estensioni dal file `debug/nucleo.py`, in modo che il debugger mostri informazioni specifiche sullo stato del nucleo. In particolare, ogni volta che il debugger riacquisisce il controllo, viene mostrato:

- lo stack delle chiamate (*backtrace*);
- 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;
- il numero di processi (utente) esistenti e le liste `esecuzione`, `pronti` (e altre liste di processi);
- alcuni dettagli sul processo attualmente in esecuzione;
- lo stato di protezione della CPU.

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

`process list`
: mostra una lista di tutti i processi attivi (utente o sistema);

`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.