Introduzione al sistema multiprogrammato

G. Lettieri

3 Aprile 2024

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 concorrentemente. Nell'esempio dell'ultima lezione abbiamo parlato di job, ma il termine che si usa oggi è *processo*.

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 Contesto

Per realizzare i processi riutilizziamo il concetto già introdotto di *contesto*. Ricordiamo che quando diciamo "testo" stiamo pensando al testo di un programma da eseguire. In un sistema multiprocesso il significato di una istruzione dipende 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.

Oltre ai contesti dei singoli processi avremo anche il "contesto privilegiato" di cui avevamo già parlato quando abbiamo introdotto la protezione. I contesti dei processi possono essere manipolati (per esempio per eseguire un cambio di contesto) solo quando il processore si trova nel contesto privilegiato. In generale, a tutte le risorse che sono condivise tra i processi, come le periferiche, si potrà accedere solo dal contesto privilegiato. In questo contesto girerà il software

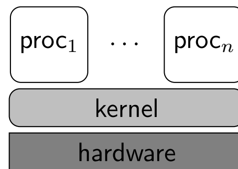

Figura 1: Rappresentazione dei livelli di privilegio e contesti.

detto *nucleo* (*kernel* in Inglese), il cui compito principale è quello di realizzare l'astrazione dei processi, e anche di fornire primitive che permettano ai processi utente di accedere alle risorse condivise in modo controllato.

La situazione a cui vogliamo arrivare è spesso rappresentata come in Figura 1. Figure di questo tipo vogliono mostrare l'esistenza di diversi contesti e i loro rapporti. I processi utente (in alto) sono meno privilegiati del "kernel" (nucleo del sistema operativo), ma non hanno in genere diverso privilegio tra essi stessi. La presenza dell'hardware in basso serve a mostrare che i processi utente devono per forza passare dal nucleo per potervi accedere, come nel nostro esempio del lettore di nastro.

Attenzione però a non farsi confondere e a non vedere in queste figure cose che non ci sono (e non possono esserci). In particolare, si potrebbe essere portati a immaginare la CPU, in quanto hardware, come posizionata "sotto" il nucleo. Questo è fuorviante, perché il software di Figura 1, sia quello utente che quello sistema, è eseguito direttamente dalla CPU e, mentre è in esecuzione, ne ha il controllo senza l'intermediazione di altro software. Per lo stesso motivo non bisogna pensare che la Figura 1 rappresenti lo stato del sistema in un qualche istante temporale: non è vero che mentre è in esecuzione proc$_1$, per esempio, abbiamo contemporaneamente, "di sotto", il nucleo che fa qualcosa (lo controlla?). La CPU può eseguire un solo software alla volta e tutto ciò che può fare e passare da una istruzione all'altra, usando i meccanismi che conosciamo (normale flusso di controllo oppure interruzioni e eccezioni). Conviene immaginare la CPU come un puntino che si muove all'interno dei rettangoli arrotondati di Figura 1, guidata dal software che sta eseguendo. Occasionalmente la CPU salta in un altro contesto (tipicamente il nucleo) per via di una interruzione o perché è caduta in una botola (*trap*). Interruzioni e trap sono i meccanismi che permettono al nucleo di riacquisire il controllo della CPU indipendemente dal volere degli utenti.

Il fatto che i processi debbano appartenere a diversi utenti non è fondamentale: in molte applicazioni uno stesso utente può voler eseguire più di un processo contemporaneamente e, anche in quel caso, vogliamo che ogni processo abbia un suo contesto indipendente (per fare in modo, per esempio, che i bug di un processo non si propaghino in altri processi). Se però uno stesso utente esegue più processi, può aver senso che questi lavorino su qualche struttura dati in comune, quindi i loro contesti potrebbero anche condividere in tutto o in parte la memoria.

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 da un bootstap loader che è in grado di interpretare i file ELF. L'output del collegatore del sistema, dunque, è direttamente utilizzabile. Sarà il modulo sistema, durante la fase di inizializzazione, a caricare gli altri due moduli (io e utente). Si veda più avanti (Sezione 2.2) per i dettagli.

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 alcuni script di utilità;
- la sottodirectory `include`, che contiene dei file `.h` inclusi dai vari sorgenti;
- la sottodirectory `build`, inizialmente vuota, destinata a contenere i moduli finiti;
- la sottodirectory `debug`, che contiene alcune estensioni per il debugger `gdb`;
- la sottodirectory `doc`, destinata a contenere la documentazione dei moduli.

La directory `nucleo-`$x.y$ contiene anche due file: 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. Il `Makefile` può essere usato per generare la documentazione lanciando il comando

```
make doc
```

Attenzione: per generare la documentazione è necessario aver installato `Doxygen` e `pandoc`. Se il comando ha successo, la documentazione si troverà in formato HTML, con l'indice in `doc/html/index.html`. Per il resto, sia il `Makefile` che lo script `run` possono essere ignorati, in quanto gli script `compile` e `boot` funzionano anche per il nucleo.[2]

2.1.1 Scrittura di programmi utente

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.

[2]Nel caso del nucleo, lo script `compile` usa internamente `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`.

```
1  #include <all.h>
2
3  void main()
4  {
5      writeconsole("Hello, world!\n", 14);
6      pause();
7      terminate_p();
8  }
```

Figura 2: Un esempio di programma utente (file utente/utente.cpp).

In Figura 2 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 7. 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 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,

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

la memoria video in modalità testo). Soli i primi 640 KiB sono liberamente utilizzabili.

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 3. Questi sono messaggi inviati sulla porta seriale della macchina virtuale. I messaggi nelle righe 1–15 arrivano dal programma `boot.bin`. Nelle righe 4–7 il programma `boot.bin` ci informa del fatto che il bootloader precedente (QEMU stesso, nel nostro caso) ha caricato in memoria tre file, e in particolare il file `build/sistema.strip`[4] all'indirizzo 0x114000. Nelle righe 8–11 ci dice come sta copiando le sezioni di questo file nella loro destinazione finale. La riga 15 ci avverte che `boot.bin` ha finito e sta per saltare all'indirizzo mostrato nella riga 12 (0x200178), dove si trova l'entry point del modulo sistema I messaggi successivi arrivano dal modulo sistema (alcuni, come quelli alle righe 45–46, arrivano dal modulo *io*). Alla riga 38 vediamo che viene inizializzato l'APIC. Le righe 19–24 contengono informazioni relative alla memoria virtuale, che per il momento ignoriamo. Di seguito viene inizializzato lo heap di sistema (riga 40, riutilizzando lo spazio occupato da `boot.bin`). Vengono poi creati i primi processi di sistema (righe 36, 37). Da questo punto in poi l'inizializzazione prosegue nel processo `main_sistema` (id 1) e `main I/O` (id 2). Il processo `main_sistema` attiva il timer alla riga 41 e crea il processo `main_io` (righe 42–43). Le righe 44–53 sono relative all'inizializzazione del modulo *io*, eseguita da questo processo. Quando il processo `main_io` termina, processo `main_sistema` crea il primo processo utente (righe 54–55) e gli cede il controllo (riga 56), semplicemente terminando (riga 57). In questo caso il processo utente esegue il codice di Figura 2, che stampa un messaggio sul video e poi termina (riga 59 di Figura 3)). Il controllo passa quindi a `dummy` (id 0), che si accorge che non ci sono più processi utente e quindi ordina lo shutdown della macchina QEMU (riga 60).

In Figura 4 mostriamo un altro esempio di programma utente, che questa volta tenta di eseguire un'azione non permessa: leggere direttamente da una porta di I/O (linea 5). 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 5). È utile imparare a leggere questi messaggi, perché contengono informazioni che aiutano a trovare più velocemente eventuali errori. La colonna centrale (che in questo caso contiene il valore 5) è l'id del processo a cui fanno riferimento questi messaggi. Il messaggio alla riga 1 contiene informazioni sull'eccezione che ha causato la terminazione forzata del

[4] Si tratta del file `build/sistema` con alcune informazioni non necessarie rimosse, in modo da occupare meno spazio in memoria.

```
INF     -       Boot loader di Calcolatori Elettronici, v1.0
INF     -       Memoria totale: 32 MiB, heap: 636 KiB
INF     -       Argomenti: /home/giuseppe/CE/lib/ce/boot.bin
INF     -       Il boot loader precedente ha caricato 3 moduli:
INF     -       - mod[0]: start=114000 end=12f580 file=build/sistema.strip
INF     -       - mod[1]: start=130000 end=1414e0 file=build/io.strip
INF     -       - mod[2]: start=142000 end=147400 file=build/utente.strip
INF     -       Copio mod[0] agli indirizzi specificati nel file ELF:
INF     -       - copiati 108560 byte da 114000 a 200000
INF     -       - copiati 970 byte da 12edb8 a 21bdb8
INF     -       - azzerati ulteriori 79030 byte
INF     -       - entry point 200178
INF     -       Creata finestra sulla memoria centrale:  [            1000,          2000000)
INF     -       Creata finestra per memory-mapped-IO:    [         2000000,        100000000)
INF     -       Attivo la modalita' a 64 bit e cedo il controllo a mod[0]...
INF     -       Nucleo di Calcolatori Elettronici, v7.1.1
INF     -       Heap del modulo sistema: [1000, a0000)
INF     -       Numero di frame: 560 (M1) 7632 (M2)
INF     -       Suddivisione della memoria virtuale:
INF     -       - sis/cond [               0,       8000000000)
INF     -       - sis/priv [      8000000000,      10000000000)
INF     -       - io /cond [     10000000000,      18000000000)
INF     -       - usr/cond [ffff800000000000, ffffc00000000000)
INF     -       - usr/priv [ffffc00000000000,                0)
INF     -       mappo il modulo I/O:
INF     -        - segmento sistema read-only  mappato a [     10000000000,      1000000f000)
INF     -        - segmento sistema read/write mappato a [     10000010000,      10000031000)
INF     -        - heap:                                 [     10000031000,      10000131000)
INF     -        - entry point: start [io.s:11]
INF     -       mappo il modulo utente:
INF     -        - segmento utente  read-only  mappato a [ffff800000000000, ffff800000005000)
INF     -        - segmento utente  read/write mappato a [ffff800000005000, ffff800000007000)
INF     -        - heap:                                 [ffff800000007000, ffff800000107000)
INF     -        - entry point: start [utente.s:10]
INF     -       Frame liberi: 7059 (M2)
INF     -       Creato il processo dummy (id = 0)
INF     -       Creato il processo main_sistema (id = 1)
INF     -       Inizializzo l'APIC
INF     -       Cedo il controllo al processo main sistema...
INF     1       Heap del modulo sistema: aggiunto [100000, 200000)
INF     1       Attivo il timer (DELAY=59659)
INF     1       Creo il processo main I/O
INF     1       proc=2 entry=start [io.s:11](1024) prio=1278 liv=0
INF     1       Attendo inizializzazione modulo I/O...
INF     2       Heap del modulo I/O: 100000B [0x10000031000, 0x10000131000)
INF     2       Inizializzo la console (kbd + vid)
INF     2       estern=3 entry=estern_kbd(int) [io.cpp:168](0) prio=1104 (tipo=50) liv=0 irq=1
INF     2       kbd: tastiera inizializzata
INF     2       vid: video inizializzato
INF     2       Inizializzo la gestione dell'hard disk
INF     2       bm: 00:01.1
INF     2       estern=4 entry=estern_hd(int) [io.cpp:509](0) prio=1120 (tipo=60) liv=0 irq=14
INF     2       Processo 2 terminato
INF     1       Creo il processo main utente
INF     1       proc=5 entry=start [utente.s:10](0) prio=1023 liv=3
INF     1       Cedo il controllo al processo main utente...
INF     1       Processo 1 terminato
INF     5       Heap del modulo utente: 100000B [0xffff800000006068, 0xffff800000106068)
INF     5       Processo 5 terminato
INF     0       Shutdown
```

Figura 3: Esempio di messaggi di log inviati sulla porta seriale.

```cpp
#include <all.h>

void main()
{
    inputb(0x60)
    pause();
    terminate_p();
}
```

Figura 4: Un esempio di programma utente che tenta di eseguire un'azione illecita.

```
 1  WRN     5       Eccezione 13 (errore di protezione), errore 0, RIP inputb [inputb.s:6]
 2  WRN     5       proc 5: corpo start [utente.s:10](0), livello UTENTE, precedenza 1023
 3  WRN     5         RIP=inputb [inputb.s:6] CPL=LIV_UTENTE
 4  WRN     5         RFLAGS=246 [-- -- -- IF -- -- ZF -- PF --, IOPL=SISTEMA]
 5  WRN     5         RAX=        fee000b0 RBX=               0 RCX=ffffffffffffe68 RDX=              60
 6  WRN     5         RDI=              60 RSI=ffffffffffffe68 RBP=ffffffffffffff0 RSP=ffffffffffffe8
 7  WRN     5         R8 =ffff800000106068 R9 =               0 R10=               0 R11=               0
 8  WRN     5         R12=               0 R13=               0 R14=               0 R15=               0
 9  WRN     5         backtrace:
10  WRN     5         > main [utente.cpp:5]
11  WRN     5       Processo 5 abortito
```

Figura 5: Esempio di messaggi di log relativi alla terminazione forzata di un processo in seguito al sollevamento di una eccezione.

processo: si tratta di una eccezione di protezione (tipo 13) e l'istruzione che l'ha generata si trova all'interno della funzione `inputb()`, alla riga 6 del file `inputb.s`. Questo è un file della `libce` (`as64/inputb.s` nella directory della libreria) e alla riga 6 troviamo appunto l'istruzione **`inb`** `%dx`, `%`**`al`**, che è vietata. La riga 2 contiene informazioni sul processo, tra cui il suo livello (UTENTE in questo caso) e la precedenza (1023 in questo caso). L'informazione più utile di questa riga è, in genere, quella che segue `corpo`: si tratta della funzione e del parametro che sono stati passati alla `activate_p()` quando questo processo è stato creato: in questo caso si tratta della funzione `start` (definita alla riga 10 di `utente.s`) con parametro 0 (mostrato tra le parentesi tonde). Le righe 3–8 mostrano anche il contenuto di tutti i registri, e in genere sono utili solo quando l'errore si è verificato in un file scritto in Assembler (come in questo caso, in cui possiamo vedere che il registro **`rdx`** conteneva 0x60 quando l'eccezione è stata sollevata). Molto più utili sono le informazioni che partono dalla riga 9: queste contengono il cosiddetto "backtrace", ovvero la pila delle chiamate di funzione ancora attive al momento dell'errore: in questo caso possiamo vedere che `inputb()` era stata chiamata dalla funzione `main`, e più precisamente alla riga 5 di `utente.cpp`, come possiamo confermare dalla Figura 4[5].

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:

[5] Il sistema invia sul log soltanto degli indirizzi numerici, che sono poi trasformati in funzione, file e numero di riga dallo script `util/show_log.pl`, usando le informazioni di debug contenute nei file della directory `build`. Per vedere il contenuto del log non processato si può usare il comando `CERAW=1 boot`.

- 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` e `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.

Si noti che il debugger è preimpostato per caricare i simboli di tutti e tre i moduli, quindi è possibile inserire breakpoint liberamente sia nel codice del modulo sistema, sia nel codice del modulo utente.