

Il Manchester Baby

G. Lettieri

A/A 2017/18

I calcolatori moderni, nell’incessante sforzo di diventare sempre più efficienti, hanno accumulato una gran quantità di meccanismi molto complessi. Allo stesso tempo, una enorme pila di software è stata interposta tra il calcolatore e l’utente finale, allo scopo di semplificargli l’utilizzo. Questi sviluppi portano fuori strada chi si accinge oggi a studiare l’architettura di un elaboratore:

- lo studente parte essendosi già formato strane idee sul funzionamento interno del calcolatore, di cui ha esperienza solo attraverso il filtro del software moderno;
- quando si confronta con la complessità dell’architettura sottostante, non ne intuisce lo scopo e finisce per perdersi nel mare dei dettagli.

Per fissare più chiaramente i principi di base, che non sono cambiati molto dalla loro introduzione negli anni ’40 del secolo scorso, può essere utile tornare, almeno brevemente, alle origini. Elimineremo in questo modo tutte le fonti di confusione che si sono accumulate nel tempo, con la speranza di poterle poi affrontare avendo le idee più chiare.

Ricordiamo che i calcolatori nascono, come dice il nome, per *calcolare*. Inizialmente si tratta di calcoli scientifici, a cui ben presto vengono affiancate elaborazioni in ambito commerciale. La caratteristica principale dei calcolatori è di essere *programmabili*: la procedura di calcolo può essere cambiata senza modificare l’hardware.

Anche se oggi usiamo queste macchine per tantissimi scopi, quando ne studiamo la struttura dobbiamo sempre tenere in mente questo: sono macchine che hanno lo scopo di eseguire programmi, e questi programmi servono a svolgere calcoli.

1 Il calcolatore SSEM

Lo Small-Scale Experimental Machine (Figura 1), anche noto come “Manchester Baby”, fu costruito all’Università di Manchester nel 1948. Si tratta del primo prototipo di calcolatore elettronico (a valvole) funzionante secondo il principio del *programma memorizzato*. Tale principio fu introdotto dal gruppo che costruì l’ENIAC negli Stati Uniti e divenne molto noto grazie a una serie di articoli scritti dal matematico John von Neumann, consulente del progetto. Anche se è quasi



Figura 1: Una replica moderna del SSEM, soprannominato “Baby”, in esposizione al *Museum of Science and Industry* di Castlefield, Manchester. (fonte: Wikipedia)

unanimemente considerato un punto di svolta nell’evoluzione dei calcolatori, il concetto di programma memorizzato non è in realtà definito precisamente. Per i nostri scopi è sufficiente dire che in un calcolatore a programma memorizzato il programma è codificato numericamente e memorizzato nella stessa memoria in cui si trovano anche i dati da elaborare.

Un simulatore Java del Baby può essere scaricato liberamente da

<http://www.davidsharp.com/baby/>.

Il Baby era soltanto un piccolo prototipo, non realmente utilizzabile per calcoli seri. Fu costruito allo scopo di testare una nuova tecnologia di memoria basata sui tubi catodici. In questa tecnologia, ormai non più in uso, i bit erano memorizzati come punti o linee su uno schermo fluorescente: venivano scritti dirigendo opportunamente un raggio catodico e riletti tramite una griglia metallica che copriva lo schermo. Questa tecnologia offre un’interessante opportunità: se il segnale che pilota il raggio catodico della memoria viene inviato anche ad un normale schermo a tubo catodico (come quello dei vecchi televisori), diventa possibile osservare direttamente il contenuto della memoria stessa. La Figura 2 mostra il dettaglio dello schermo utilizzato proprio a questo scopo nel Baby. La Figura 3, invece, mostra lo schermo del Baby riprodotto sul simulatore. La memoria consiste soltanto di 32 locazioni, ciascuna di 32 bit. Una locazione occupa una intera riga della matrice. Ogni locazione è identificata da un numero da 0 a 31: quella più in alto è la 0 e le altre sono numerate a seguire. Questi

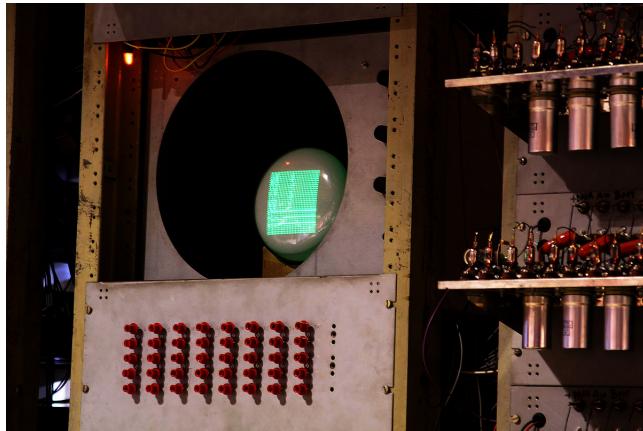


Figura 2: L'output su CRT (*Cathode Ray Tube*, tubo catodico). Lo schermo mostra il contenuto della memoria come una matrice di 32×32 punti o linee (fonte: Wikipedia)

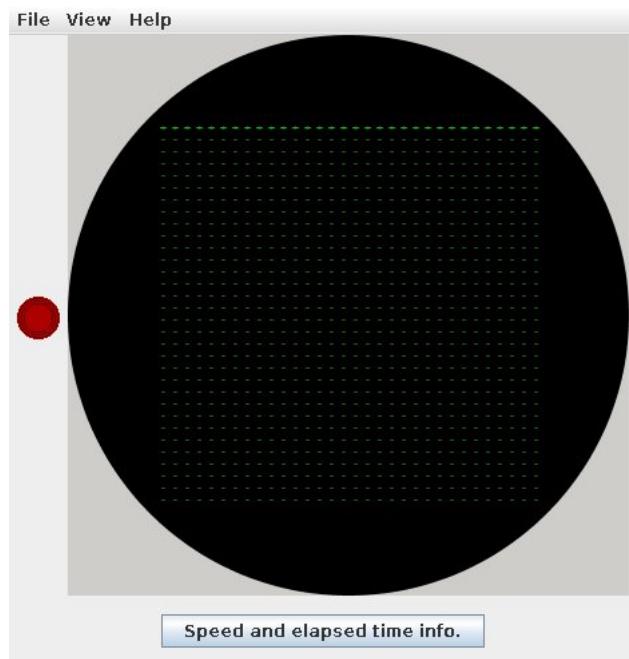


Figura 3: La memoria della SSEM come mostrata dal simulatore.

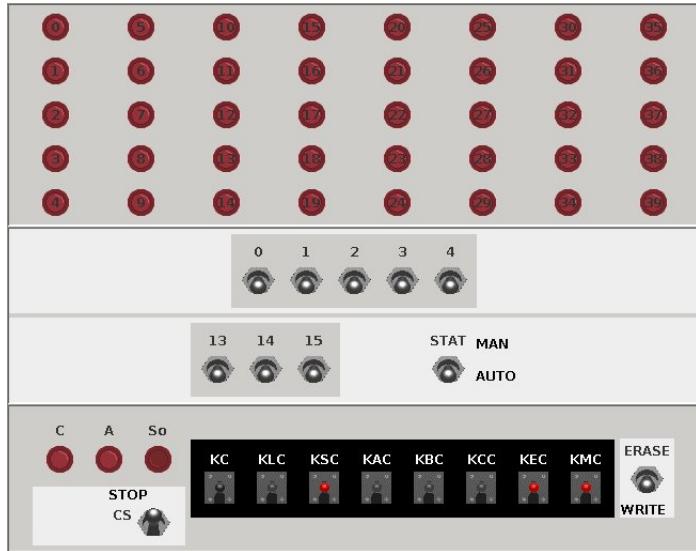


Figura 4: Il pannello di controllo della SSEM nel simulatore.

identificatori, già allora chiamati *indirizzi*, non sono visibili da nessuna parte. Il dispositivo di memoria, però, dato un indirizzo, è in grado di selezionare la corrispondente riga, permettendo di leggerne il contenuto (converire i corrispondenti punti e linee in impulsi elettrici) o cambiarne il contenuto (convertire impulsi elettrici in punti e linee).

Ogni locazione può memorizzare indifferentemente un numero intero (rappresentato in complemento a 2 su 32 bit) oppure una istruzione. Niente, nella memoria, permette di distinguere un numero da una istruzione: il significato dei bit dipende solo dall'uso che se ne fa. Tutto ciò è vero anche per le memorie moderne. Anzi, l'immagine di Figura 3 è un buon modo di visualizzare la memoria di un qualunque calcolatore. Su una sola cosa la memoria del Baby è un po' particolare: in ogni locazione, il bit meno significativo si trova a sinistra.

L'ingresso/uscita della macchina è ridotto al minimo. Le uniche uscite sono lo schermo stesso e la lampadina visibile in alto a sinistra in Figura 2 rappresentata in rosso in Figura 3. Lo schermo permette di vedere tutto il contenuto della memoria (e anche dei registri, come vedremo), mentre la lampadina si accende quando la macchina si ferma, presumibilmente perché il programma è terminato. L'unico dispositivo di ingresso è dato dal pannello visibile parzialmente in Figura 2, subito sotto lo schermo, e mostrato in Figura 4 come riprodotto nel simulatore. A parte alcuni interruttori per cancellare la memoria e i registri e avviare o fermare l'esecuzione, notiamo le cinque file di bottoni rossi in alto e i cinque interruttori numerati da 0 a 4 subito sotto. Questi 5 interruttori permettono di selezionare una riga qualsiasi della memoria scrivendone il numero in binario. Anche qui il bit meno significativo va inserito agendo sull'interruttore più a sinistra (quello numerato con 0). Se la levetta in basso a destra è impostata

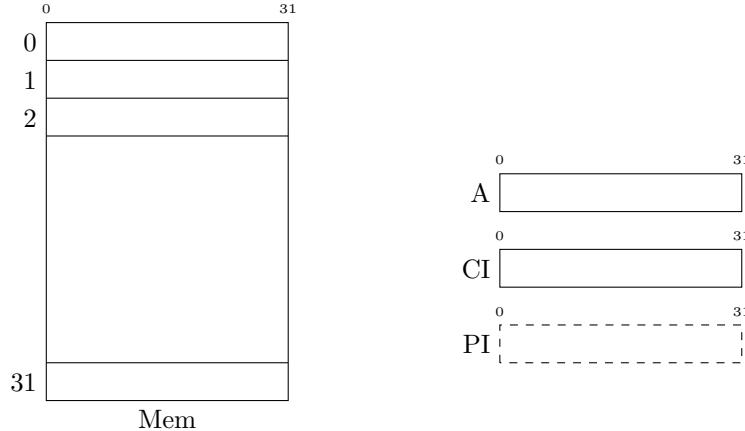


Figura 5: La memoria (a sinistra) e i registri (a destra) del SSEM.

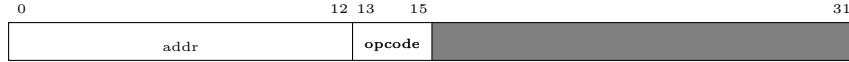
su WRITE, i primi 32 bottoni rossi permettono di accendere il corrispondente bit della riga selezionata (i bottoni da 33 a 39 non servono a niente). Se la levetta è impostata su ERASE, gli stessi bottoni permettono invece di azzerare il corrispondente bit.

Il fatto che il Baby fosse solo un piccolo prototipo lo rende ideale per il nostro esempio, in quanto ci permette di eliminare un gran numero di distrazioni: abbiamo tutta la memoria sott'occhio e possiamo direttamente accendere o spegnere tutti i bit che vogliamo. Per quanto riguarda la memoria, dunque, non ci sono segreti.

Il Baby è ridotto all'osso anche per quanto riguarda il supporto alla programmazione. Le Figure 5 e 6 mostrano tutto ciò che il programmatore deve sapere per programmare la macchina (le informazioni equivalenti per un moderno processore Intel si trovano in una serie di volumi di quasi 4000 pagine in totale). La Figura 5 mostra la memoria, che già conosciamo, e i tre registri disponibili: un accumulatore (A), il contatore di programma (CI) e un registro non utilizzabile direttamente dal programmatore, destinato a contenere l'istruzione che la macchina sta eseguendo (PI, *Present Instruction*). I registri sono realizzati con la stessa tecnologia della memoria, anche se consistono ciascuno di un'unica riga. I bottoni C, A e Sc in basso a sinistra in Figura 4 permettono di vedere sullo schermo il contenuto di CI e PI (bottone C), di A (bottone A) o della memoria (bottone So).

Quando l'interruttore STOP/CS è impostato su CS, la macchina:

1. incrementa CI di 1;
2. legge dalla memoria la locazione di indirizzo CI e la copia in PI;
3. esegue l'istruzione contenuta in PI;



opcode	mnemonico	effetto
000	JMP	$CI \leftarrow Mem[addr]$
100	JRP	$CI \leftarrow CI + Mem[addr]$
010	LDN	$A \leftarrow -Mem[addr]$
110	STO	$Mem[addr] \leftarrow A$
-01	SUB	$A \leftarrow A - Mem[addr]$
011	CMP	se $A < 0$, $CI \leftarrow CI + 1$
111	STP	halt

Figura 6: Il formato e l'insieme delle istruzioni del SSEM.

4. se l'istruzione non era di stop, torna al punto 1; altrimenti, accende la lampadina e non fa altro.

Se una istruzione non modifica il contenuto di CI (al punto 3), la macchina esegue in sequenza le istruzioni contenute in memoria, grazie all'auto-incremento di CI (punto 1). La macchina è un po' particolare, dal momento che l'incremento di CI avviene in ogni caso, prima di prelevare l'istruzione. In particolare, se all'accensione CI contiene zero, la prima istruzione che verrà eseguita sarà quella che si trova all'indirizzo 1. A parte alcuni dettagli (come quest'ultimo), un processore moderno si comporta essenzialmente nello stesso modo del processore del Baby, da quando lo accendiamo a quando lo spegniamo.

Le possibili istruzioni sono solo 7 e sono mostrate in Figura 6. In alto in figura è mostrato anche il *formato* delle istruzioni, cioè come queste devono essere codificate in memoria affinché la macchina sia in grado di interpretarle. Abbiamo detto che ogni istruzione occupa una riga (32 bit), ma in realtà i 16 bit più significativi sono ignorati. Dei 16 bit meno significativi, i primi 12 possono contenere un indirizzo di memoria (*addr*). Dal momento che la memoria è di sole 32 locazioni, solo i primi 5 bit di *addr* sono realmente utilizzati e gli altri sono ignorati. I bit 13, 14 e 15 identificano l'istruzione da eseguire (*opcode*, codice operativo). L'istruzione 010 (LDN) legge la cella di memoria di indirizzo *addr* e la copia nel registro *A*. Si noti che ne cambia anche il segno, che è un'altra peculiarità di questa macchina. L'operazione 110 fa la copia nell'altro verso (ma senza cambiare il segno). I codici 001 e 101 sono utilizzati per la stessa istruzione, che esegue una sottrazione tra il contenuto di *A* e il contenuto della locazione di memoria puntata da *addr*, scrivendo il risultato in *A*. Si noti che le istruzioni permettono di scrivere o leggere solo intere locazioni (interi righe, quindi). In particolare, non esistono i byte.

istr.	commento
LDN X	carichiamo X , ottenendo $-x$ nell'accumulatore
CMP	se $-x$ è negativo, x era positivo e non dobbiamo fare niente
STO X	altrimenti sostituiamo x con $-x$
STP	fermiamo la macchina

Figura 7: Un esempio di programma. La locazione X deve contenere un valore (x) che alla fine deve essere sostituito dal suo valore assoluto.

Le istruzioni 000, 100, 011 e 111 permettono di alterare il flusso di esecuzione delle istruzioni. L'istruzione 111 semplicemente lo arresta, in modo che l'utente possa osservare il risultato del programma. Le istruzioni 000 e 100 scrivono un nuovo valore in CI, e sono dunque dei salti incondizionati: 000 è assoluto e 100 è relativo al valore corrente di CI. L'istruzione 011 è molto importante: decide di saltare o meno l'istruzione successiva in base al segno del numero contenuto in A. La possibilità di eseguire azioni diverse in base al risultato di elaborazioni precedenti è ciò che permette alla macchina di eseguire potenzialmente qualunque algoritmo (avendo a disposizione sufficiente memoria). Anche se le istruzioni del Baby hanno alcune peculiarità (la decisione di avere solo un caricamento con cambio di segno, il salto condizionale che permette di saltare una sola istruzione), sono comunque rappresentative del tipo di istruzioni che si trovano comunemente in tutti i processori.

Non c'è altro da sapere sul processore del Baby, e dunque anche qui non ci sono segreti. Dato lo stato iniziale della memoria, tutto ciò che la macchina farà è completamente determinato.

2 Un esempio di programmazione

Programmare la macchina significa decidere lo stato iniziale della memoria, che conterrà sia il programma, sia i dati che il programma dovrà elaborare.

Proviamo a scrivere un programma molto semplice. Supponiamo di avere un numero x , memorizzato in una locazione X . Vogliamo scrivere un programma che sostituisca x con il suo valore assoluto.

Il dato da elaborare, in questo caso, è il solo numero x . Il programma può essere scritto in tanti modi. Un modo è il seguente: se carichiamo x nell'accumulatore, il Baby cambierà anche il segno. Se abbiamo ottenuto un numero negativo, vuol dire che x era positivo, e dunque già uguale al suo valore assoluto. Se invece otteniamo un numero positivo, vuol dire sia che x era negativo, sia che ora abbiamo il suo valore assoluto nell'accumulatore. Ci basta dunque scrivere il contenuto dell'accumulatore all'indirizzo X , e abbiamo terminato.

Questa idea si traduce nel programma mostrato in Figura 7

Per fare in modo che la macchina lo esegua, dobbiamo fare tre cose:

- decidere dove mettere il programma e i dati all'interno della memoria;

2. tradurre tutto (programma e dati) in binario;
3. inizializzare la memoria con i bit ottenuti al passo precedente.

Occupiamoci del punto 1. Ricordiamo che il processore eseguirà per prima l'istruzione che si trova all'indirizzo 1, e quindi procederà automaticamente agli indirizzi successivi, se non incontra istruzioni di salto. Conviene dunque caricare il nostro programma a partire dall'indirizzo 1 fino all'indirizzo 4. Il nostro programma opera su un solo dato, il numero che si trova alla locazione X . Dobbiamo decidere dove allocare questo dato. Le locazioni di memoria sono tutte equivalenti, quindi possiamo sceglierne una qualunque, con l'unico vincolo di non sceglierne una già usata per qualcos'altro. Nel nostro caso le uniche locazioni già occupate saranno quelle da 1 a 4. Scegliamo, per esempio, $X = 20$. All'indirizzo 20 dovremo caricare il numero x di cui vogliamo calcolare il valore assoluto. Il nostro programma dovrebbe poter funzionare con un qualunque x , ma per eseguirlo dobbiamo sceglierne uno. Scegliamo, per esempio, $x = -1$.

Ora passiamo al punto 2. Per tradurre il programma in binario, esaminiamo ogni istruzione utilizzando la tabella di Figura 6. La prima istruzione da convertire è “LDN 20”. In binario, 20 è 10100_2 ($16 + 4$), e dunque i bit 0–4 dell'istruzione dovranno valere 00101 (si ricordi che in questa macchina il bit meno significativo va a sinistra, quindi il numero appare al contrario rispetto a come siamo abituati). Dalla tabella di Figura 6 vediamo che il campo *opcode* (bit 13–15) deve valere 010. Gli altri bit della riga verranno ignorati e potremmo assegnarvi un valore qualsiasi, ma non abbiamo motivo per assegnarvi un valore diverso da 0. Si passa poi all'istruzione “CMP”, che non usa il suo campo *addr*, quindi dobbiamo preoccuparci solo dei bit 13–15, che devono valere 011. Procediamo così anche per le altre due istruzioni, ottenendo la tabella seguente:

riga	0	1	2	3	4	13	14	15
1	0	0	1	0	1	0	1	0
2						0	1	1
3	0	0	1	0	1	1	1	0
4						1	1	1

Dobbiamo anche convertire il numero x . Il valore -1 , rappresentato in complemento a 2 su 32 bit corrisponde a 32 bit tutti pari a 1.

Terminiamo infine con il punto 3. Per caricare la riga 1 nella memoria del Baby, supponendo che inizialmente sia completamente azzerata, selezioniamo la riga 1 tramite le levette centrali (levetta 0 in basso, le altre in alto). Quindi, con la levetta “ERASE/WRITE” impostata su “WRITE”, premiamo i bottoni rossi numero 1, 3 e 14. Procediamo così per tutte le altre righe, ottenendo la configurazione di memoria illustrata in Figura 8.

A questo punto possiamo avviare l'esecuzione: portiamo tutte le levette 0–4 e 13–14 in basso e poi abbassiamo anche la levetta “STOP/CS”. Se non abbiamo commesso errori, si accenderà la lampada di stop e alla locazione 20 dovremmo osservare il valore 1 (un solo bit acceso a sinistra) al posto di -1 .

Figura 8: Programma di Figura 7 caricato in memoria. Abbiamo scelto $X = 20$ e $x = -1$.

Questa laboriosa procedura deve essere eseguita in modo sostanzialmente equivalente anche quando si programma un calcolatore moderno. La differenza principale consiste nel fatto che possiamo utilizzare dei programmi già pronti (assemblatore, collegatore e caricatore) per automatizzare le varie fasi.

In genere questi programmi sono eseguiti sul calcolatore stesso su cui poi verrà eseguito anche il programma finale. Questa circostanza, sicuramente molto comoda, genera però gran confusione negli studenti alle prime armi, che sono indotti a pensare che l'assemblatore (o, più tipicamente, il compilatore) continua a fare qualcosa anche mentre il programma finale è in esecuzione, oppure che per eseguire un programma sia necessaria la contemporanea presenza di tutti questi strumenti. L'unica cosa che conta, invece, ai fini dell'esecuzione di un programma, è come è stata inizializzata la memoria al suo avvio (Figura 8). Per avere le idee un po' più chiare è molto meglio immaginare che tutta l'operazione di traduzione e caricamento venga svolta a mano, come abbiamo fatto ora, anche quando usiamo compilatori, assemblatori, collegatori, caricatori e persino ambienti integrati di sviluppo.

Calcolatori Elettronici: introduzione

G. Lettieri

1 Marzo 2022

1 Introduzione e richiami

La Figura 1 mostra l’architettura generale di un calcolatore moderno. Nel corso approfondiremo ogni componente. I principali argomenti di cui parleremo sono:

- protezione;
- interruzioni;
- memoria virtuale.

Questi tre argomenti ci permetteranno di parlare della *multiprogrammazione*: come eseguire “contemporaneamente” più programmi su un sistema che ha un solo processore. Studieremo tutti i dettagli che ci permetteranno di realizzare un (semplificato, ma funzionante) *nucleo di sistema operativo multiprogrammato*. Raffineremo inoltre l’architettura dell’elaboratore parlando di cache, DMA (Direct Memory Access), bus PCI. Infine, esamineremo la struttura interna di un processore moderno in grado di eseguire le istruzioni in parallelo, fuori ordine e speculativamente.

Se escludiamo questi argomenti avanzati, l’architettura di base di un calcolatore moderno resta sorprendentemente simile a quella del Manchester Baby che abbiamo visto nella lezione introduttiva. Esamineremo di seguito i componenti principali (che comunque dovrebbero essere tutti già noti da corsi precedenti).

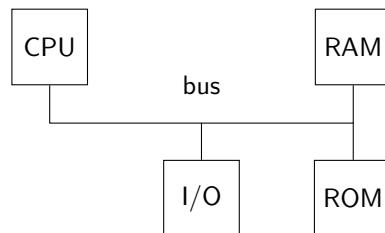


Figura 1: Architettura generale di un calcolatore moderno.

1.1 Il bus (senza DMA)

Il bus connette tra loro i vari dispositivi in modo che ciascuno possa comunicare con ciascun altro. La comunicazione deve avvenire sempre tramite operazioni di lettura o scrittura. Entrambi i tipi di operazione richiedono che venga specificato un “indirizzo”. Gli indirizzi sono normalmente dei numeri in sequenza. Il nome indirizzo fu scelto per ricordare gli indirizzi delle case. La metafora vuole che immaginiamo una lunghissima via in cui ci sono case in cui abitano numeri, ciascuna con il proprio indirizzo. L’operazione di lettura all’indirizzo x chiede chi abita all’indirizzo x . L’operazione di scrittura del numero y all’indirizzo x fa sì che ora y abiti all’indirizzo x (scacciando il precedente abitante). Si noti che sia gli indirizzi che gli abitanti sono numeri: è importante non confondere le due cose.

Ogni operazione è richiesta da un componente ed eseguita da un altro. Nel Manchester Baby il bus connette solo la CPU e la memoria, mentre l’I/O è sostanzialmente diverso da quello che si trova in un sistema moderno, come vedremo più avanti. La Fig. 1 mostra invece una architettura più regolare, basta su un bus comune che permette a qualunque componente di richiedere una operazione a qualunque altro. Questo diventerà importante quando parleremo di DMA, ma per il momento le uniche combinazioni possibili sono le seguenti:

1. la CPU legge o scrive sulla memoria;
2. la CPU legge o scrive sull’I/O.

Le operazioni devono essere eseguite una per volta. In ogni operazione è l’indirizzo che permette di capire se è coinvolta la memoria o l’I/O. Il discriminante può essere operato o riservando indirizzi diversi alla memoria e all’I/O, oppure definendo *spazi di indirizzamento* separati (vie diverse, nella metafora degli indirizzi delle case). Nel secondo caso, ogni operazione deve specificare anche lo spazio di indirizzamento coinvolto (memoria o I/O). Ogni operazione è vista da tutti i dispositivi collegati al bus e ciascuno di essi deve autonomamente capire se l’operazione è rivolta a lui oppure no. Per farlo deve confrontare l’indirizzo dell’operazione con gli indirizzi a lui riservati. Se l’operazione non è rivolta a lui, deve ignorarla. Se nessun dispositivo riconosce l’indirizzo possono succedere cose diverse in dipendenza dal tipo di bus: noi faremo l’ipotesi che le operazioni vengano comunque completate, con le scritture che non hanno effetto e le letture che restituiscono un valore casuale.

1.2 La memoria

L’idea è esattamente la stessa che per il Manchester Baby. Si tratta di un sistema organizzato in “celle”, ciascuna composta di un certo numero di bit che è uguale per tutte le celle. Una volta che la memoria è montata sul bus, ogni cella ha un indirizzo unico che la identifica. La differenza più grande rispetto al Baby (oltre al numero di celle) è che la memoria di quest’ultimo era organizzata in celle

relativamente grandi (dette in seguito “parole”), in particolare di dimensione uguale a quella dei registri. Come vedremo, le memorie moderne continuano ad essere accessibili in questo modo, ma permettono anche di accedere a unità più piccole. In particolare, nelle memorie moderne ogni singolo *byte* ha il proprio indirizzo distinto.

Per il resto, i seguenti punti continuano a essere veri:

- ogni cella contiene più precisamente una sequenza di bit; questa può essere sempre interpretata (da chi usa la memoria) come un numero naturale espresso in base due, ma il significato del contenuto di una cella dipende esclusivamente dall’uso che se ne fa;
- la memoria sa eseguire soltanto due tipi di operazioni: lettura e scrittura;
- sa eseguire una sola operazione per volta;
- per eseguire una lettura si deve specificare l’indirizzo della cella che si vuole leggere (la memoria risponde con il contenuto della cella);
- per eseguire una scrittura si deve specificare l’indirizzo della cella che si vuole scrivere, e il nuovo contenuto della cella (la memoria sostituisce il vecchio contenuto con il nuovo);
- la memoria è completamente passiva; non cambia il suo stato se non quando qualcuno ordina una operazione di scrittura;
- *tutte* le celle della memoria contengono *sempre* qualcosa, anche prima di eseguire qualunque scrittura: questo perché ogni bit è memorizzato da un dispositivo che può assumere solo uno tra due stati (0 o 1); all’accensione ci saranno zeri e uno a caso;
- ciascuna operazione richiede un tempo all’incirca costante, indipendentemente da quali altre operazioni sono state richieste precedentemente (memoria ad accesso casuale).
- nella memoria non c’è scritto cosa significhino i vari numeri, se rappresentino istruzioni, caratteri, o qualunque altra cosa.

1.3 I/O (senza interruzioni e DMA)

Anche se i dispositivi di ingresso/uscita sono tanti e vari, tutte le possibili interazioni sono trasformate in operazioni di lettura o scrittura. Queste operazioni coinvolgono particolari celle, dette registri o porte di I/O. Ogni dispositivo (tastiera, stampante, ...) sarà collegato al sistema tramite una *interfaccia*, all’interno della quale si trovano i registri. Indipendentemente dall’interfaccia a cui appartiene, nel momento in cui l’interfaccia è montata sul bus ogni suo registro acquisisce un indirizzo che lo identifica univocamente, esattamente come le celle della memoria. Le operazioni di lettura e scrittura sui registri sono del tutto simili alle analoghe operazioni sulla memoria. La differenza è che ciascuna

operazione può avere effetti collaterali, come causare la stampa di un carattere sulla carta di una stampante. Anche la lettura I/O di un registro può avere effetti collaterali, come cambiare lo stato interno di una periferica. Per esempio, leggere il codice dell'ultimo tasto premuto dal registro di ingresso della tastiera fa sì che la tastiera smetta di memorizzare quel codice; la prossima volta che il registro verrà letto, la tastiera restituirà il codice del *nuovo* ultimo tasto premuto (se ve ne sono). Ciò è completamente diverso dalla memoria, in cui il contenuto delle celle cambia solo se vi si scrive e due operazioni di lettura consecutive restituiranno sempre lo stesso valore. I dispositivi di I/O, inoltre, possono cambiare il loro stato interno in seguito alla loro interazione con il mondo esterno al calcolatore (per esempio, la tastiera memorizza il codice di un nuovo tasto se qualcuno lo preme).

Dall'interno del calcolatore, il mutamento di stato di una periferica è visibile esclusivamente dal fatto che cambia il contenuto dei registri della sua interfaccia. Le interfacce, inoltre, (in assenza dei meccanismi di interruzione e DMA) aspettano passivamente che qualcuno legga il nuovo valore dei registri. Questa è anche la principale differenza con il Manchester Baby, che abbiamo visto nell'esempio introduttivo. Nel Manchester Baby l'I/O è *direttamente* collegato alla RAM¹. Ricordiamo che l'I/O del Baby consiste nello schermo, che ci permette di vedere direttamente il contenuto della RAM, e della pulsantiera che ci permette di scrivere direttamente in RAM. Con “direttamente” intendiamo che non è necessaria (e, in questo caso, neanche possibile) la mediazione del software: la pulsantiera, per esempio, funziona appena si accende la macchina, quando in memoria non c’è ancora alcun programma. Inoltre, nessuno ci impedisce di premere i tasti *mentre* sta girando il programma, cambiando così il contenuto della memoria in modi molto difficili da controllare (in pratica si settano i bit della riga che in quel momento il programma stava leggendo o scrivendo). In un calcolatore moderno questo non è possibile: tutto deve essere sotto il controllo del programma, compresi di dispositivi di I/O.

1.4 La CPU (senza interruzioni e protezione)

Se escludiamo i meccanismi avanzati delle interruzioni e della protezione, quello che resta è funzionalmente molto simile alla CPU del Baby. In particolare, la CPU è un sistema che sa eseguire una sequenza di *istruzioni*. Le istruzioni operano sullo *stato* della CPU e/o interagiscono con l'esterno ordinando operazioni di lettura o scrittura sul bus. Lo stato della CPU è contenuto in un insieme di registri. I registri sono funzionalmente simili alle celle della memoria, con la differenza che si trovano all'interno della CPU. Le istruzioni sono codificate in numeri e contenute in memoria. Uno dei registri della CPU contiene l'indirizzo della prossima istruzione da eseguire (Instruction Pointer, IP). La CPU, da quando la accendiamo a quando la spegniamo, fa solo ed esclusivamente le seguenti cose:

¹L'I/O è anche collegato direttamente alla CPU, ma possiamo tralasciare questo ulteriore dettaglio.

1. preleva dalla memoria l'istruzione puntata dall'IP;
2. la decodifica, la esegue e aggiorna l'IP;
3. torna al punto 1.

Il modo in cui viene aggiornato l'IP al punto 2 dipende dall'istruzione eseguita. Per la maggior parte delle istruzioni, l'IP viene semplicemente incrementato in modo che punti all'istruzione che segue in memoria, secondo l'ordine degli indirizzi. Un programma, dunque, è per lo più una sequenza di azioni da eseguire una dopo l'altra (come la parola “programma” normalmente indica in altri contesti). Alcune istruzioni possono cambiare l'IP per eseguire dei salti. L'istruzione `hlt` ferma il processore (possiamo pensare che non modifichi l'IP).

Punti (tutti già noti) da tenere a mente:

1. tutto ciò che il processore fa, lo fa perché sta prelevando o eseguendo una istruzione;
2. tranne che quando esegue `hlt`, il processore non smette mai di eseguire istruzioni;
3. le istruzioni sono quelle del linguaggio macchina del processore; il processore non è in grado di eseguire nient'altro;
4. tutto quello che vede il processore è il suo stato corrente e l'istruzione da eseguire; il processore non ricorda le istruzioni passate e non si aspetta particolari istruzioni future:
 - tutto ciò che resta di una istruzione alla fine della sua esecuzione è l'effetto che essa ha avuto sullo stato dei registri del processore, delle celle di memoria, o sui registri di I/O;
 - dualmente, tutto ciò che una istruzione vede quando comincia la sua esecuzione è ciò che si trova nei registri del processore, nelle celle di memoria o nei registri di I/O.

1.5 La memoria ROM

Il fatto che, in un calcolatore moderno, l'ingresso/uscita non funziona direttamente ma deve essere comandato da un programma, crea un problema di *bootstrap*: come facciamo a caricare un programma in memoria se l'unica via di ingresso ha essa stessa bisogno di che ci sia un programma in memoria per poter funzionare? La soluzione è di avere una memoria non volatile che contenga già un programma, fisso, che ha lo scopo di caricare il programma vero e proprio da qualche periferica di I/O (hard disk, rete, DVD, etc.).

1.6 Il flusso di controllo

Quanto esposto sopra è tutto ciò che l'hardware sa fare. Tutto ciò che un calcolatore fa, per quanto complicato possa apparire, deve ridursi a questo. È il software (il programma contenuto in memoria ed eseguito dalla CPU) a orchestrare questi comportamenti elementari in modo da ottenerne di più complessi.

Tutta l'architettura esiste solo per eseguire il software, ed è il software che comanda. La CPU è sotto il controllo del software: la CPU deve eseguire l'istruzione corrente (quella il cui indirizzo si trova nel suo instruction pointer), ed è inoltre l'istruzione stessa a dire quale deve essere l'istruzione successiva. Le istruzioni che non sono di salto lo dicono implicitamente (l'istruzione successiva è la prossima in memoria), mentre quelle di salto lo dicono esplicitamente. L'unico altro meccanismo che dice al processore qual è l'istruzione successiva è quello delle interruzioni, che per il momento ignoriamo.

Il controllo “fluisce” dunque da una istruzione all'altra come deciso dal software stesso. Se il software è composto di più parti, come per esempio più subroutine o diverse applicazioni o librerie, è sempre una sola di queste parti per volta che ha il controllo del processore. Il controllo può essere solo “palleggiato”: quando una subroutine S_1 ne invoca un'altra S_2 (per es. tramite una istruzione **call**) si dice che S_1 *trasferisce il controllo* a S_2 . A questo punto il processore obbedisce a S_2 e non più a S_1 , fino a quando S_2 non deciderà di *restituire il controllo* a S_1 (per es. eseguendo una istruzione **ret**).

1.7 Hardware e software

Alcuni trovano difficoltà nel capire se una certa cosa è fatta in hardware o in software, e in generale a capire “chi” svolge determinate azioni.

Per orientarsi può essere utile tenere sempre a mente cosa i vari componenti *sanno*. In particolare, si rifletta sul fatto che la CPU sa solo ciò che è contenuto nei suoi registri e, in particolare, vede del programma solo l'istruzione che di volta in volta sta eseguendo, senza ricordare le istruzioni passate e senza guardare quali siano le istruzioni future (che pure sono già scritte in memoria). Questo limita fortemente le cose che la CPU può fare. Nello scenario di S_1 che trasferisce il controllo a S_2 , per esempio, la CPU non può controllare che S_2 ritorni a S_1 : non sa cosa S_2 stia facendo e, con la visione limitata di una istruzione alla volta, non è in grado di capirlo. La CPU di cui ci occupiamo noi, per di più, non sa nemmeno che nel passato era stata eseguita una **call** e che dunque prima o poi deve essere eseguita una **ret**.²

Non si deve però giungere alla conclusione che dunque tutto è fatto in software. Quando si ritiene che una certa operazione x sia fatta in software, si deve essere in grado di rispondere alle seguenti domande:

²Verso la fine del corso vedremo una CPU più intelligente che cerca di ricordare e prevedere più cose, ma lo fa al solo scopo di andare più veloce, restando per il resto equivalente a quella stupida.

- se x è fatta in software, vuol dire che da qualche parte in memoria ci deve essere un programma p (anche di una sola istruzione) che fa x ; so scrivere questo programma?
- il software non ha effetto se la CPU non lo esegue; come fa il flusso di controllo a passare da p quando serve che sia fatta x ?

Se anche una di queste domande risulta assurda, è probabile che x non sia fatta in software.

L'architettura Intel/AMD a 64 bit

G. Lettieri

1 Marzo 2022

1 Registri e istruzioni

I processori Intel/AMD a 64 bit sono una evoluzione dei precedenti processori a 32 bit, evoluzione a loro volta dei precedenti a 16 bit.

Lo *stato* dell'elaboratore è dato dal contenuto dei registri del processore, dal contenuto della memoria e dal contenuto dei registri di I/O. Vediamoli di seguito.

All'interno del processore si trovano 16 registri di uso generale, i cui nomi sono mostrati in Figura 1 insieme al registro **rip**, che è l'instruction pointer, e al registro **rflags**, che è il registro dei flag. Tutti i registri sono grandi 64 bit. Ci sono anche altri registri di controllo, che vedremo successivamente.

È possibile anche riferire solo alcune parti di ogni registro, utilizzando un nome diverso, secondo la Tabella 1. In particolare, è possibile riferire i 32 bit meno significativi, i 16 bit meno significativi e gli 8 bit meno significativi. Per i soli registri **rax**, **rbx**, **rcx** ed **rdx** è possibile anche riferire gli 8 bit successivi a **al**, **bl**, **cl** e **dl**, utilizzando i nomi nell'ultima colonna della Tabella 1. Tuttavia ci sono delle complicate limitazioni sul loro utilizzo, e conviene ignorare l'esistenza di questi ulteriori sotto-registri.

Per quanto riguarda lo spazio di memoria, il processore può potenzialmente riferire 2^{64} byte distinti utilizzando indirizzi di 64 bit. In pratica, però, quasi tutti i processori Intel/AMD ad oggi disponibili limitano i bit realmente utilizzabili a 48, ottenendo uno spazio di memoria di 2^{48} B = 256 TiB¹, che è comunque molto grande. I più recenti offrono l'opzione di utilizzare 57 bit, corrispondenti ad uno spazio di memoria di 2^{57} B = 128 PiB. I 48 (o 57) bit liberamente utilizzabili sono quelli meno significativi. Gli altri 16 (o 7) devono essere tutti uguali al bit numero 47 (o 56). Questo vuol dire che lo spazio di memoria si presenta come in Figura 2: si possono indirizzare soltanto due porzioni contigue, ciascuna grande 2^{47} (caso con 48 bit, Fig. 2a) o 2^{56} byte (caso con 57 bit, Fig. 2b), una all'inizio e l'altra alla fine dello spazio degli indirizzi. Gli indirizzi che rispettano questa regola sono detti in *forma canonica*. È il processore stesso a generare un errore se si tenta di utilizzare un indirizzo che non è in forma canonica.

Lo spazio di I/O, infine, è costituito da 2^{16} locazioni di un byte (64 KiB).

¹B sta per byte mentre Ki = 1024, Mi = 1024², Gi = 1024³, Ti = 1024⁴ e Pi = 1024⁵.

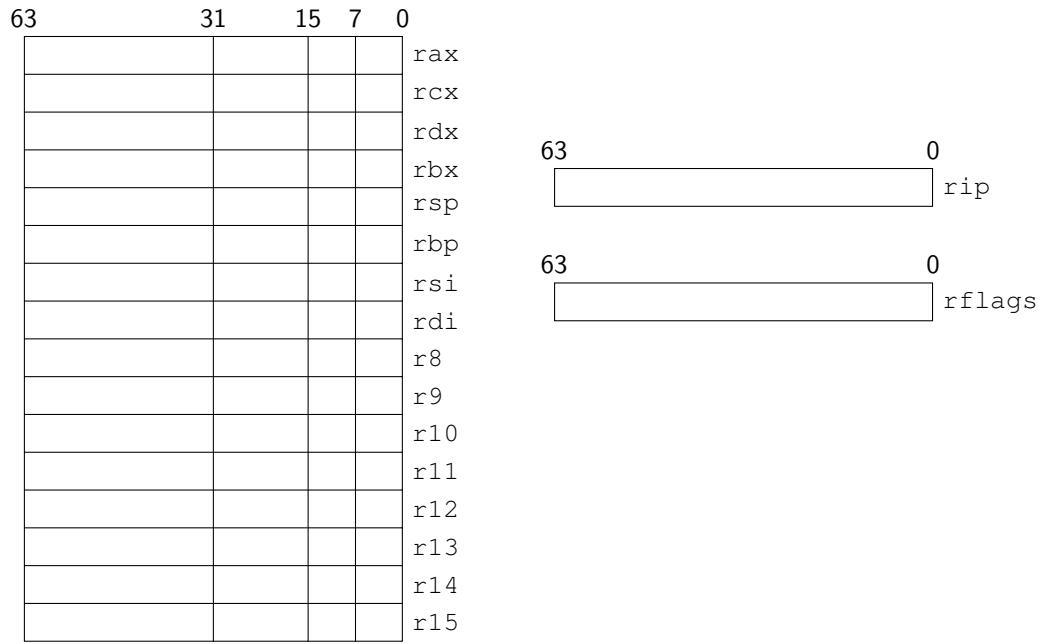


Figura 1: I registri del processore.

64b	32b	16b	8b	8b
rax	eax	ax	al	ah
rcx	ecx	cx	cl	ch
rdx	edx	dx	dl	dh
rbx	ebx	bx	bl	bh
rsp	esp	sp	spl	
rbp	ebp	bi	tpl	
rsi	esi	si	sil	
rdi	edi	di	dil	
r8	r8d	r8w	r8b	
r9	r9d	r9w	r9b	
r10	r10d	r10w	r10b	
r11	r11d	r11w	r11b	
r12	r12d	r12w	r12b	
r13	r13d	r13w	r13b	
r14	r14d	r14w	r14b	
r15	r15d	r15w	r15b	

Tabella 1: Nomi dei sotto-registri.



Figura 2: Spazio di indirizzamento di memoria (non in scala).

Le istruzioni riconosciute dal processore prevedono normalmente due operandi di ingresso e sovrascrivono il risultato sul secondo operando. La sintassi assembler è la seguente:

codice-operativo primo-operando, secondo-operando

Sono possibili tre modalità di indirizzamento degli operandi:

- **immediato:** l'operando è una costante contenuta nell'istruzione stessa (al massimo su 32 bit); in assembler l'operando deve essere preceduto dal carattere \$;
- **registro:** l'operando è contenuto in uno dei registri del processore; in assembler si usa il carattere % seguito dal nome del registro;
- **memoria:** l'operando è contenuto in memoria; l'istruzione deve dunque specificarne l'indirizzo.

Il caso di indirizzamento di memoria è il più complicato, in quanto è possibile chiedere al processore di *calcolare* l'indirizzo richiesto (ricordarsi che gli indirizzi sono numeri). Ci sono due modalità principali di calcolo dell'indirizzo, con vari sotto casi:

- La modalità con spiazzamento, base, indice e scala, che usa la seguente sintassi:

spiazzamento (base, indice, scala)

dove:

- *spiazzamento* è una costante (8 o 32 bit, con segno)
- *base* è il nome di un registro a 64b;
- *indice* è il nome di un registro a 64b;
- *scala* può valere 1, 2, 4 oppure 8.

Il processore calcolerà l'indirizzo sommando lo *spiazzamento* (esteso con segno a 64b), il contenuto del registro *base* e il contenuto del registro *indice* moltiplicato per la *scala*.

Ci sono vari casi particolari, vediamo i più comuni:

- se lo spiazzamento è zero, si può omettere;
- se la scala è 1, si può omettere insieme alla seconda virgola;
- si può omettere tutta la parte tra parentesi; in questo caso l'indirizzo coincide con lo spiazzamento e l'indirizzamento è detto *diretto*;
- si possono omettere l'indice e la scala, con le relative virgolette; in questo caso l'indirizzo è ottenuto sommando lo spiazzamento (esteso con segno a 64b) e il contenuto del registro base; se si omette anche lo spiazzamento, l'indirizzamento è detto *indiretto*.

- Relativa a **rip**, con la seguente sintassi:

spiazzamento (%rip)

dove *spiazzamento* è una costante (8 o 32 bit, con segno). Il processore calcolerà l'indirizzo sommando *spiazzamento* (esteso con segno a 64b) al contenuto di **rip**.

Si noti che il secondo operando non può essere di tipo immediato e che al più un operando può essere di tipo memoria.

Solo l'istruzione movabs ammette operandi immediati di 64b e spiazzamenti di 64b, e solo nei seguenti formati:

- movabs *\$costante, %registro*: copia la *costante* nel *registro*;
- movabs *spiazzamento, %rax*: leggi dall'indirizzo *spiazzamento* in memoria e scrivi in **rax** (o nei suoi sotto-registri).
- movabs *%rax, spiazzamento*: scrivi il contenuto di **rax** (o dei suoi sotto-registr) all'indirizzo *spiazzamento* in memoria.

Ogni istruzione può lavorare con operandi di 8, 16, 32 o 64 bit. Se uno dei due operandi è un registro l’assemblatore può dedurre automaticamente la dimensione, altrimenti è necessario specificarla aggiungendo uno dei seguenti suffissi al codice operativo dell’istruzione:

- b per operandi di tipo byte;
- w per operandi di tipo “parola” (*word*, due byte).
- l per operandi di tipo “parola lunga” (*long word*, quattro byte).
- q per operandi di tipo “parola quadrupla” (*quad word*, otto byte).

Si consiglia di aggiungere sempre il suffisso, anche quando non serve.

2 Esempio di programma

Prepariamo un semplice esempio di programma da fare eseguire al nostro elaboratore. Dato che ci interessa principalmente sapere cosa succede durante l’esecuzione all’interno dell’elaboratore, usiamo il linguaggio assembler, che è il più vicino al linguaggio macchina: ogni istruzione di assembler si traduce in una istruzione di linguaggio macchina. Sempre per lo stesso motivo, ci conviene pensare che tutta la procedura di preparazione del programma in linguaggio macchina (scrittura del file sorgente, assemblamento, collegamento) sia svolta all’esterno del calcolatore, su un altro calcolatore o in qualunque altro modo, anche se in pratica usiamo lo stesso calcolatore per fare tutto. L’esecuzione del nostro programma comincia dal momento in cui esso è stato caricato in memoria; a quel punto, esiste solo il linguaggio macchina e tutta la procedura precedente non conta più.

La procedura inizia preparando un file di testo che contiene il programma in linguaggio assembler. Si consideri il file di Figura 3. I numeri di riga servono per riferimento e non fanno parte del contenuto del file. L’assemblatore produce una o più sequenze di byte da caricare in memoria, in base alle direttive o alle istruzioni contenute nel file sorgente. Le parole chiave che cominciano per “.” sono *direttive* e servono a chiedere all’assemblatore di svolgere vari compiti. Alla riga 1 troviamo la direttiva **.data** che chiede all’assemblatore di aggiungere alla *sezione data* ciò che segue nel file fino alla prossima direttiva che specifica una nuova sezione (in questo caso, la direttiva **.text** alla riga 8). Le sezioni sono sequenze di byte che possono essere caricate indipendentemente. La sezione “data” verrà caricata in una zona di memoria accessibile sia in lettura che in scrittura, mentre la sezione “text” in una zona di sola lettura. Normalmente la sezione data contiene variabili, mentre la sezione text contiene codice, ma niente di tutto ciò è imposto o controllato dall’assemblatore.

Alla riga 2 troviamo la definizione di una *etichetta* (num1 in questo caso). Le etichette servono a dare un nome all’indirizzo del primo byte che le segue. Abbiamo bisogno delle etichette perché, avendo delegato all’assemblatore e al collegatore il compito di decidere dove caricare le sezioni, non sappiamo gli

```
1 .data
2 num1:
3     .quad    0x1122334455667788
4 num2:
5     .quad    0x9900aabbcdddeeff
6 risu:
7     .quad    -1
8 .text
9 .globl _start, start
10 start:
11 _start:
12     movabsq $num1, %rax
13     movq (%rax), %rcx
14     movabsq $num2, %rax
15     movq (%rax), %rbx
16     addq %rbx, %rcx
17     movabsq $risu, %rax
18     movq %rcx, (%rax)
19
20     movq $13, %rbx
21     mov $1, %rax
22     int    $0x80
```

Figura 3: Un esempio di programma scritto in Assembler GNU per x86_64.

indirizzi delle entità che definiamo. Tramite le etichette possiamo riferirci a tali indirizzi simbolicamente, e lasciare che siano poi l’assemblatore e il collegatore a sostituirle con i veri indirizzi.

Alla riga 3 troviamo la direttiva **.quad** seguita da un numero. La direttiva chiede all’assemblatore di riservare 8 byte (a partire dal punto della sezione in cui è arrivato) e di inizializzarli con il numero specificato. Possiamo specificare il numero in varie basi (esadecimale, in questo caso): l’assemblatore provvederà a convertire in ogni caso il numero in binario. Più in generale è possibile riservare e inizializzare una sequenza di quad, semplicemente facendo seguire **.quad** da una lista di numeri separati da virgole. Altre direttive di questo tipo sono **.byte**, per riservare e inizializzare una sequenza di byte, **.long** (per doppie parole) o **.word** (parole).

L’effetto delle righe 2 e 3 è di allocare e inizializzare una variabile che occupa una parola quadrupla e darle il nome `num1`. Le righe 4–5 e 5–6 fanno la stessa cosa con `num2` e `risu`.

Alla riga 8 diciamo all’assemblatore che quanto segue deve essere aggiunto alla sezione “text”.

Alla riga 9 troviamo la direttiva **.global** seguita dalle etichette `_start` e `start`. Con questa direttiva stiamo chiedendo di rendere queste etichette visibili al collegatore. Il collegatore cercherà una di queste due per sapere qual è l’indirizzo della prima istruzione del programma (alcuni collegatori cercano `start` e altri `_start`, e per questo e le definiamo entrambe; se ci si limita a Linux è sufficiente `_start`). Il registro **rip** verrà inizializzato con questo indirizzo quando il programma dovrà essere eseguito.

Alle righe 10 e 11 definiamo le etichette `start` e `_start`. Si noti che non c’è differenza sintattica tra la definizione di `num1`, `num2` e `risu` da una parte e `_start` (o `start`) dall’altra, anche se nelle nostre intenzioni le prime sono variabili mentre `_start` è un’etichetta del programma. Non è casuale, in quanto per l’assemblatore non c’è alcuna differenza tra queste etichette. In ogni caso l’etichetta serve a dare un nome all’indirizzo del byte che la segue. L’assemblatore non segnalerà alcun errore se proviamo a saltare a `num1` o se proviamo a leggere o scrivere all’indirizzo `_start`.

Alle righe 12–22 c’è il programma vero e proprio. Ogni riga contiene una istruzione per il processore. Quello che l’assemblatore farà sarà di tradurre ogni riga nella corrispondente sequenza di byte di linguaggio macchina, andando così a formare la sezione `text`. Si noti che questo non è molto diverso da quanto l’assemblatore fa con la direttiva **.quad** (o **.byte**, etc.): si tratta in ogni caso di costruire a poco a poco la sequenza dei byte che compongono la sezione che poi verrà caricata in memoria. Di fatto, niente vieta di usare le direttive **.quad** etc. nella sezione `text` o di scrivere una istruzione nella sezione `data`. In ogni caso si ottengono dei byte, e questi non sono di per sé né dati, né istruzioni: è solo nel momento in cui li utilizziamo che vengono interpretati in un modo o in un altro.

Il programma vuole calcolare la somma dei due numeri memorizzati agli indirizzi `num1` e `num2` e scrivere il risultato all’indirizzo `risu`. Per farlo, carica il primo numero nel registro **rcx** (righe 12–13), il secondo numero nel registro

rbx (righe 14–15), li somma scrivendo il risultato in **rcx** (riga 16), infine copia il risultato in **risu** (righe 17–18). Le righe 20–22 servono a dire al sistema operativo che il programma è terminato e per il momento conviene ignorarle.

Osserviamo la riga 12. L'obiettivo è di caricare l'indirizzo della prima variabile in **rax**, in modo da poter poi caricare il primo numero tramite indirizzamento indiretto (riga 13). Il primo operando è di tipo *immediato* (lo si riconosce dal carattere \$). Non ci si lasci confondere dall'uso dell'etichetta: l'etichetta sparirà e verrà sostituita dal suo valore numerico (si ricordi che gli indirizzi sono numeri). L'istruzione sta caricando una costante in **rax**. Questa costante (avendo usato l'etichetta num1) non è altro che l'indirizzo del primo numero da sommare. Stiamo usando movabs perché, non sapendo dove il programma verrà caricato, non possiamo sapere se tale indirizzo può essere contenuto in soli 32 bit, e movabs è l'unica istruzione che accetta operandi immediati a 64 bit.

Si noti che al posto delle istruzioni 12 e 13 avremmo potuto scrivere

```
movabsq num1, %rax  
movq %rax, %rcx
```

In questo caso il primo operando della movabs è di tipo memoria (lo si riconosce dal fatto che non inizia né con \$, né con %). L'istruzione sta ordinando al processore di eseguire una operazione di lettura in memoria di 8 byte (dal momento che la destinazione è **rax**) a partire dall'indirizzo num1 (di nuovo: nel linguaggio macchina finale num1 sparirà e al suo posto ci sarà il vero indirizzo). Questa istruzione può caricare solo in **rax**, quindi abbiamo bisogno della successiva per copiare il valore letto in **rcx**, dove lo volevamo.

Stesse considerazioni valgono per le righe 14–15 e, nella direzione opposta, per le righe 17–18.

Supponiamo che questo file si chiami **sum.s**. Per assemblarlo lanciamo il comando

```
as sum.s
```

Se non ci sono errori di sintassi, l'assemblatore non stampa niente e produce il file **a.out** (assembler output). È possibile cambiare il nome del file prodotto usando l'opzione **-o** seguita dal nome desiderato. Conviene farlo, visto che anche il collegatore usa il nome **a.out** come default:

```
as sum.s -o sum.o -g
```

(Abbiamo aggiunto anche l'opzione **-g**, che include nel file le informazioni utilizzate dal debugger.)

Tale file deve essere poi passato al collegatore per produrre l'eseguibile:

```
ld sum.o -o sum -g
```

(Anche qui abbiamo usato l'opzione **-o** per specificare il nome dell'eseguibile e l'opzione **-g** per il debugger.)

```

sum-example/sum.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <_start>:
 0: 48 b8 00 00 00 00 00  movabs $0x0,%rax
 7: 00 00 00
 a: 48 8b 08          mov    (%rax),%rcx
 d: 48 b8 00 00 00 00 00  movabs $0x0,%rax
14: 00 00 00
17: 48 8b 18          mov    (%rax),%rbx
1a: 48 01 d9          add    %rbx,%rcx
1d: 48 b8 00 00 00 00 00  movabs $0x0,%rax
24: 00 00 00
27: 48 89 08          mov    %rcx,(%rax)
2a: 48 c7 c3 0d 00 00 00  mov    $0xd,%rbx
31: 48 c7 c0 01 00 00 00  mov    $0x1,%rax
38: cd 80              int   $0x80

```

Figura 4: Output del comando `objdump -d sum`.

Possiamo esaminare il prodotto dell’assemblatore e del collegatore con il comando `objdump`. Per esempio, possiamo chiedere di vedere il codice macchina prodotto dall’assemblatore nella sezione `text`:

```
objdump -d sum.o
```

In questo caso dovremmo ottenere un output simile a quello di Figura 4. Le righe a partire da quella che inizia con “0:” mostrano il contenuto della sezione `text`. Ogni riga mostra un offset all’interno della sezione, seguita da una sequenza di byte che si trovano a partire da quell’offset (tutti i numeri sono in esadecimale). Sull’estrema destra tali byte sono interpretati come istruzioni di assembler. Si noti che questa interpretazione, che è l’operazione inversa rispetto a quanto fa l’assemblatore, è fatta da `objdump` senza guardare il file sorgente. Si noti come `num1`, `num2` e `risu` sono state sostituite con zero. Questo perché neanche l’assemblatore sa quanto valgono, in quanto è solo il collegatore che decide dove le varie sezioni dovranno essere caricate.

Per vedere il risultato prodotto dal collegatore (sempre nella sezione `text`), scriviamo

```
objdump -d sum
```

L’output è mostrato in Figura 5 e si interpreta in modo simile a quello di Figura 4. In questo caso, però, il numero all’inizio di ogni riga rappresenta l’*indirizzo* (scelto dal collegatore) a partire dal quale verranno caricati i byte

```

sum-example/sum:      file format elf64-x86-64

Disassembly of section .text:

0000000000401000 <_start>:
401000: 48 b8 00 20 40 00 00  movabs $0x402000,%rax
401007: 00 00 00
40100a: 48 8b 08          mov     (%rax),%rcx
40100d: 48 b8 08 20 40 00 00  movabs $0x402008,%rax
401014: 00 00 00
401017: 48 8b 18          mov     (%rax),%rbx
40101a: 48 01 d9          add    %rbx,%rcx
40101d: 48 b8 10 20 40 00 00  movabs $0x402010,%rax
401024: 00 00 00
401027: 48 89 08          mov     %rcx,(%rax)
40102a: 48 c7 c3 0d 00 00 00  mov     $0xd,%rbx
401031: 48 c7 c0 01 00 00 00  mov     $0x1,%rax
401038: cd 80              int    $0x80

```

Figura 5: Output del comando objdump -d sum.

mostrati in quella riga. Qui vediamo che num1 etc. hanno assunto il loro valore finale. Si noti che il nostro programma consiste dei byte che si trovano nella parte centrale dell'output: questo è tutto quello che il processore vedrà quando il programma sarà caricato e messo in esecuzione.

Il programma può essere infine caricato ed eseguito scrivendo

```
./sum
```

In questo caso il programma non produce alcun output, perché ci siamo limitati a scrivere il risultato in memoria, senza inviarlo ad alcuna periferica di I/O. Possiamo però vederlo in funzione utilizzando il debugger. Scriviamo

```
gdb sum
```

gdb è un debugger a riga di comando, per il quale esistono varie interfacce grafiche (per esempio, ddd). Possiede una semplice interfaccia semi-grafica incorporata, che si può far partire premendo prima i tasti “Ctrl+x” e poi il tasto “a”. Per mettere in funzione il nostro programma settiamo prima un breakpoint all'etichetta _start scrivendo b _start (invio), e poi facciamo partire l'esecuzione scrivendo r (e poi invio). Possiamo far avanzare il programma di una istruzione alla volta premendo n (invio). Per esaminare il contenuto dei registri possiamo cambiare il layout dell'interfaccia con il comando layout reg (invio). In qualunque momento possiamo terminare il debugger scrivendo q

(invio). `gdb` ha molti comandi, si consiglia di consultare qualche guida in rete o l'help incorporato (comando `help`).

Calcolatori Elettronici: indirizzi e oggetti

G. Lettieri

3 Marzo 2019

Gli indirizzi sono relativi ad un bus: tutti i componenti collegati al bus, in grado di rispondere a richieste di lettura o scrittura, devono avere degli indirizzi assegnati in modo univoco. I possibili indirizzi vanno praticamente sempre da 0 fino ad un massimo della forma $2^n - 1$ per qualche n che dipende dal bus. Questo perché gli indirizzi viaggiano su un numero prefissato di piedini, fili o tracce, ciascuna delle quali può assumere solo i valori 0 o 1. Se queste tracce sono n , il numero totale di indirizzi possibili è dunque 2^n .

Gli indirizzi esistono sempre tutti, nel senso che è sempre possibile richiedere una lettura o una scrittura a qualunque indirizzo, anche se l'indirizzo non è assegnato a nessun componente (il risultato di una tale richiesta dipende dal bus; come abbiamo detto, assumiamo per ora che le scritture non abbiano effetto e le letture restituiscano un valore casuale).

Per questi motivi, il modo più naturale di rappresentare gli indirizzi di un bus è come la sequenza di tutti i numeri binari, senza segno, su n bit. Conviene acquisire familiarità con le rappresentazioni dei numeri in base 16 e 8, in quanto queste basi permettono di rappresentare gli indirizzi in forma molto più compatta e, allo stesso tempo, facilmente convertibile da e verso la base 2. Alcuni numeri molto frequenti devono subito richiamare alla mente la loro rappresentazione nelle varie basi: in base 2, 2^a è un 1 seguito da a zeri mentre $2^a - 1$ è composto da a 1. Se a è un multiplo di 4, allora 2^a in base 16 è 1 seguito da $a/4$ zeri, mentre $2^a - 1$ è rappresentato come $a/4$ cifre F (questo perché ogni cifra esadecimale corrisponde a 4 cifre binarie). Similmente per la base 8: se a è multiplo di 3, allora 2^a è 1 seguito a $a/3$ zeri e $2^a - 1$ è composto da $a/3$ cifre 7. In generale, conviene usare queste basi ogni volta che dobbiamo ragionare sulla struttura binaria di un qualche valore. È bene familiarizzarsi con i seguenti casi notevoli:

- un byte corrisponde a 2 cifre esadecimali;
- 512 corrisponde a 1000 in base 8;
- 4 Ki corrisponde a 1000 in base 16 e 1.0000 in base 8;
- 1 Mi corrisponde a 10.0000 in base 16;
- 4 Gi corrisponde a 1.0000.0000 in base 16.

Le operazioni sugli indirizzi (come aggiungere una costante a un indirizzo, o sottrarre due indirizzi) vanno sempre pensate come operazioni modulo 2^n . Questo comporta che l'indirizzo che precede l'indirizzo 0 è l'indirizzo $2^n - 1$, e l'indirizzo successivo a $2^n - 1$ è l'indirizzo zero.

1 Scostamenti (*offset*)

Dati due indirizzi x e y su n bit, possiamo chiederci quanto y si discosta da x calcolando il valore $y - x$ modulo 2^n , detto *offset* di y rispetto a x .

In ogni caso, l'offset conta il numero di indirizzi che è necessario “saltare”, partendo da x , per raggiungere y . In particolare, l'offset di x rispetto a se stesso è zero. Gli offset possono essere anche negativi: il loro valore assoluto rappresenta comunque il numero di indirizzi da saltare partendo da x per raggiungere y , ma andando nella direzione degli indirizzi decrescenti. Per esempio, l'offset -1 rappresenta l'indirizzo che precede x (nel caso $x = 0$ si ricordi che l'indirizzo precedente è $2^n - 1$).

Se rappresentiamo gli offset come numeri in complemento a 2 su n bit, diventa indifferente considerarli con o senza segno. Facciamo un esempio con $n = 4$. Gli indirizzi vanno dunque da 0 a 15. L'offset -1 si rappresenta come 1111 in complemento a 2 su 4 bit. Prendiamo $x = 3$. Se interpretiamo 1111 come numero con segno, l'offset è -1 e saltando un indirizzo all'indietro arriviamo a $y = 2$. Se ora interpretiamo l'offset 1111 come il numero senza segno 15, dobbiamo saltare 15 indirizzi in avanti partendo da $x = 3$. Dopo averne saltati 12 arriviamo all'indirizzo 15, saltandone un altro ripartiamo dall'indirizzo 0 e con gli ultimi due salti arriviamo a $y = 2$, come prima. Si noti, però, che l'equivalenza vale solo se l'offset è rappresentato su un mero di bit maggiore o uguale n . Se è rappresentato su meno bit, è necessario sapere se va interpretato come numero con o senza segno.

2 Intervalli (*range*)

Un *intervallo* è una sequenza di indirizzi. Conviene quasi sempre rappresentarlo specificando il primo indirizzo che fa parte dell'intervallo, sia x , e il primo, successivo a x , che *non* ne fa parte, sia y . In altre parole, come un intervallo di numeri aperto a destra $\{n \mid x \leq n < y\}$, che si rappresenta più concisamente come $[x, y)$. Si noti che, in base a questa definizione, un intervallo $[x, y)$ in cui $y \leq x$ è vuoto. Ci limitiamo a considerare intervalli $[x, y)$ in cui $y \geq x$ ¹

Uno dei vantaggi di questa scelta è che l'offset tra y e x (in altre parole, $y - x$) rappresenta sempre il numero di indirizzi che fanno parte dell'intervallo. Questo vale anche per l'intervallo $[x, x)$, che è vuoto. Se avessimo scelto di includere anche l'estremo di destra avremmo dovuto invece sommare 1 alla differenza dei due estremi.

¹Per semplicità, dunque, evitiamo di considerare intervalli che attraversano l'ultimo indirizzo rappresentabile e ripartono da zero. Questi avrebbero $y < x$.

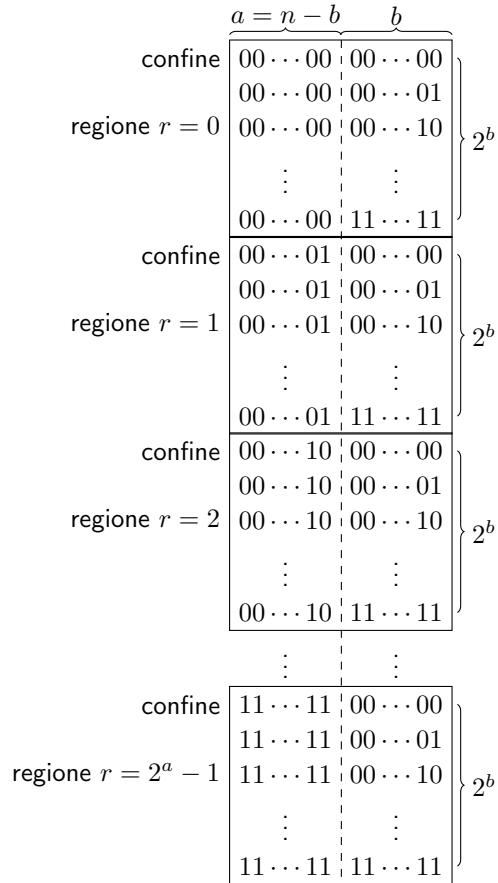


Figura 1: Scomposizione degli indirizzi in regioni naturali di 2^b .

L'indirizzo x è detto *base* dell'intervallo. Se di un intervallo conosciamo la base x e la lunghezza l , l'intervallo è dato da $[x, x + l)$. Si noti ancora l'assenza di fastidiosi -1 .

D'altra parte, nella notazione aperta a destra bisogna ricordarsi che l'ultimo indirizzo che *fa parte* dell'intervallo $[x, y)$ è $y - 1$ (l'indirizzo che precede y , che è il primo che non ne fa parte).

3 Confini (*boundaries*) e regioni naturali

Dato un qualunque $b \leq n$, tutti gli indirizzi multipli di 2^b sono detti *confini* di 2^b . I confini di 2^b terminano con almeno b zeri nella loro rappresentazione in base 2.

Si noti che l'indirizzo zero è un confine per qualunque b . Tutti i confini si trovano quindi partendo da 0 e procedendo ad offset successivi di 2^b . Questi confini delimitano degli intervalli di indirizzi che chiamiamo *regioni naturali* di 2^b . Ometteremo l'aggettivo “naturali” quando non si crea ambiguità. Useremo il termine regione (naturale) in modo generico, ma in molti casi introdurremo dei nomi particolari per regioni di particolare interesse (per esempio, *cacheline*, *pagina*, ...). Le regioni sono in totale 2^{n-b} , o 2^a se poniamo $a = n - b$ (Figura 1). Incidentalmente, notiamo che se un indirizzo x è un confine per 2^b è anche un confine per qualunque $b' < b$.

Possiamo assegnare ad ogni regione un numero progressivo r compreso tra 0 e $2^a - 1$. Il numero r , detto *numero di regione*, serve ad identificare univocamente ogni regione. Si noti che tutti (e soli) gli indirizzi che fanno parte della stessa regione contengono negli a bit più significativi proprio il numero della regione a cui appartengono. Invece, i b bit meno significativi contengono l'offset dell'indirizzo rispetto alla base della regione (il confine), detto *offset all'interno della regione*. Ogni indirizzo è dunque identificato dal numero di regione e dal suo offset (all'interno della regione). Ovviamente, lo stesso indirizzo può essere identificato in modi diversi in base alla dimensione della regione scelta.

Dato un numero b e un indirizzo x su n bit è dunque immediato, in hardware, trovare il suo numero di regione (di 2^b) e il suo offset: gli $a = n - b$ bit più significativi x danno il numero di regione, e i b bit meno significativi danno l'offset. Supponiamo ora di voler fare la stessa cosa in software, supponendo di avere una variable **unsigned long** x che contiene un indirizzo, e di conoscere b . Prendiamo come n la dimensione in bit di un **unsigned long**, che nel nostro caso è 64. Per ottenere il numero di regione possiamo scrivere semplicemente

```
r = x >> b; // numero di regione
```

Per ottenere l'offset usiamo l'AND bit a bit con una maschera che ha b cifre meno significative ad 1 e tutte le altre a zero:

```
o = x & ((1UL << b) - 1); // offset
```

I caratteri UL che seguono la costante 1 servono a specificarne il tipo come Unsigned Long. Si ricordi che per default le costanti numeriche del C++ hanno tipo **int**.

Vediamo anche come si ottiene l'*indirizzo* della regione a cui x appartiene. Per indirizzo della regione intendiamo il confine da cui parte. Possiamo ottenere questo indirizzo con una maschera che ha a bit significativi a 1 e gli altri a 0. Questa maschera non è altro che il NOT della maschera che abbiamo usato prima:

```
c = x & ~((1UL << b) - 1); // confine
```

Dato un intervallo $[x, y]$, vogliamo spesso sapere quali sono la prima e l'ultima regione toccate dall'intervallo. Per farlo è sufficiente operare come prima, usando gli indirizzi x (prima toccata) e $y - 1$ (ultima toccata).

4 Oggetti e allineamenti

Un *oggetto* è una sequenza di un certo numero di byte, sia l . Possiamo assegnare indirizzi a tutti i byte di un oggetto scegliendo l'indirizzo del suo primo byte e assegnando gli altri indirizzi in sequenza. Se x è l'indirizzo del primo byte dell'oggetto, questo viene ad occupare l'intervallo $[x, x + l)$. L'indirizzo x è quasi unanimemente considerato l'indirizzo dell'oggetto stesso (e non solo del suo primo byte) e anche noi faremo così.

Diamo un po' di definizioni:

- si dice che un oggetto o è *allineato a 2^b* (sottendendo byte) se il suo indirizzo è un confine di 2^b ;
- se la dimensione di o' è una potenza di 2, si dice che o è *allineato a o'* se è allineato alla dimensione di o' ;
- infine, se la dimensione di o è una potenza di 2, si dice che o è *allineato naturalmente* se è allineato a se stesso.

Un oggetto allineato naturalmente occupa interamente una regione della sua dimensione.

Si noti che l'allineamento non è una proprietà dell'oggetto, ma dell'indirizzo che gli è stato assegnato. Lo stesso oggetto può essere allineato a dimensioni diverse in base a dove si trova.

Calcolatori Elettronici: la memoria centrale

G. Lettieri

4 Marzo 2022

Vogliamo vedere come organizzare e collegare al bus una memoria che permetta l'accesso sia alle parole di 2, 4 o 8 byte.

Il byte rappresenta l'unità di indirizzamento: non è possibile leggere o scrivere una unità più piccola di un byte in una singola operazione di lettura o scrittura. Se, per esempio, è necessario modificare un singolo bit all'interno di un byte, è necessario leggere l'intero byte, modificare il bit lasciando gli altri inalterati, quindi riscrivere il nuovo byte.

Dal momento che lavoriamo con parole di più byte, ci possiamo chiedere in che ordine i byte della parola si trovino in memoria. Architetture diverse possono utilizzare ordini diversi. Nel nostro caso l'architettura usa l'ordinamento detto *little endian*: il byte meno significativo si trova all'indirizzo più piccolo, seguito dagli altri byte in ordine di significatività. Per esempio, supponiamo di avere una parola quadrupla in memoria, a partire da un indirizzo i , che memorizza il numero esadecimale 0x1122334455667788. Il byte di indirizzo i conterrà 0x88, il byte di indirizzo $i + 1$ conterrà 0x77, e così via fino al byte di indirizzo $i + 7$ che conterrà 0x11. Un altro ordinamento molto utilizzato (per esempio da noi stessi quando scriviamo i numeri su un foglio) è quello *big endian*¹, che consiste nello scrivere il numero partendo dalla parte più significativa. In Fig. 1 abbiamo ordinato i byte di ogni riga da destra verso sinistra proprio per ovviare alla differenza tra il modo in cui l'architettura AMD64 memorizza le parole e il modo in cui noi siamo abituati a leggerle.

Possiamo realizzare un modulo di memoria che possa anche leggere o scrivere più di un byte in una singola operazione e nello stesso tempo impiegato per un singolo byte. Nel nostro caso il modulo può leggere o scrivere una intera parola di 64 bit (8 byte contigui) purché questa sia *allineata naturalmente*. Il nostro modulo di memoria può anche leggere o scrivere, in una sola operazione, una parola di 32 bit (4 byte) o una parola di 16 bit (2 byte), purché queste non attraversino i confini di 8 byte (cioè siano interamente contenute in una regione naturale di 8 byte).

Per visualizzare più facilmente queste limitazioni conviene rappresentare lo spazio di indirizzamento di memoria non come una sequenza di indirizzi di byte, ma come una sequenza di *righe* o *linee* di 8 indirizzi di byte (il numero massimo

¹I nomi *little endian* e *big endian* vengono indirettamente da *I viaggi di Gulliver* di Jonathan Swift, grazie a un famoso articolo del 1980 reperibile a questo indirizzo: <https://www.ietf.org/rfc/ien/ien137.txt>

numero di riga	+7	+6	+5	+4	+3	+2	+1	+0	indirizzo di riga
0									0
1									8
2									16
3									24
								...	
$2^{61} - 1$									$2^{64} - 8$

Figura 1: Organizzazione dello spazio di memoria su righe di 64 bit. Ogni casella rappresenta un byte. I numeri $+0, +1, \dots$ sono gli offset all'interno della riga.

di byte che può essere trasferito in una singola operazione da un modulo di memoria). Si veda la Fig. 1, dove le righe sono disposte in orizzontale. In questo modo i dati accessibili in un'unica operazione saranno sempre contenuti in una singola riga. In pratica adottiamo una scomposizione degli indirizzi in regioni di 2^8 e disegniamo le regioni in orizzontale.

Si noti che per avere le parole da 2 byte e le parole da 4 byte sempre contenute in una riga è sufficiente che anch'esse siano allineate naturalmente, ma non è necessario. Per esempio, una parola da 4 byte che inizi in una riga alla colonna $+3$ sarebbe interamente contenuta nella riga, pur non essendo allineata naturalmente (per essere allineata naturalmente dovrebbe iniziare alla colonna $+0$ o alla colonna $+4$).

Nel seguito assumeremo che gli indirizzi della CPU e del BUS siano su n bit. Non assumiamo che n sia 64, in quanto questo è solo il valore massimo teorico dell'architettura Intel/AMD64. I processori di questa famiglia usciti fino ad ora hanno, di fatto, un numero inferiore di bit di indirizzo (per esempio, 36) e altre considerazioni, che vedremo, limitano il massimo numero di bit dei processori futuri della stessa famiglia a un valore comunque inferiore a 64.

Per specificare completamente una richiesta di operazione di lettura la CPU deve presentare alla memoria l'indirizzo del primo byte e il numero di byte (per una operazione di scrittura dovremo anche presentare il nuovo contenuto dei byte in questione). Queste due informazioni vengono specificate in modo indiretto nel seguente modo:

1. la CPU scomponete l'indirizzo del primo byte in numero di riga e offset all'interno della riga;
2. la CPU e il bus prevedono $n - 3$ fili, che chiamiamo $A\{n - 1\} - A3$, destinati a trasportare soltanto il numero di riga;
3. al posto dell'offset del primo byte (che sarebbe stato contenuto nei fili $A2$, $A1$ e $A0$) e del numero di byte sono previsti 8 fili di *byte enable*, uno per ogni byte della riga selezionata da $A\{n - 1\} - A3$.

Lo scopo delle linee di byte enable, che chiamiamo /BE7-/BE0, è di selezionare singolarmente i byte della riga che la CPU intende leggere (o scrivere) nell'operazione.

Esempi:

- Supponiamo che la CPU stia eseguendo una istruzione **movq** 512, %**rax**. Deve dunque ordinare una operazione di lettura di una parola da 8 byte all'indirizzo 512. Si tratta della riga n. $512/8 = 64$, che va dagli indirizzi 512 a 519. Avremo $A\{n-1\} = A\{n-2\} = \dots = A10 = 0$, $A9 = 1$, $A8 = A7 = \dots = A3 = 0$, in quanto 512 è 1000000000 in binario, e /BE7 = /BE6 = … = /BE0 = 0 (tutti i byte abilitati);
- Supponiamo invece che l'istruzione sia **movl** 512, %**eax**. Si tratta ora di una lettura di una parola da 4 byte all'indirizzo 512: avremo $A\{n-1\}$ –A3 come prima, ma /BE7 = /BE6 = /BE5 = /BE4 = 1 (byte 7–4 disabilitati) e /BE3 = /BE2 = /BE1 = /BE0 = 0 (byte 3–0 abilitati);
- Consideriamo ora **movl** 516, %**eax**. Ora la lettura coinvolge una parola da 4 byte all'indirizzo 516: avremo $A\{n-1\}$ –A3 ancora come prima, in quanto siamo sempre all'interno della stessa riga. I byte enable saranno però diversi: /BE7 = /BE6 = /BE5 = /BE4 = 0 (byte 7–4 abilitati) e /BE3 = /BE2 = /BE1 = /BE0 = 1 (byte 3–0 disabilitati).

Si noti che questo tipo di codifica permetterebbe di specificare molti più casi di quelli che ci servono. Per esempio, con /BE7 = /BE5 = /BE3 = /BE1 = 1 e /BE6 = /BE4 = /BE2 = /BE0 = 0 potremmo leggere solo i byte di indirizzo pari all'interno della riga selezionata. Di fatto tali possibilità non sono sfruttate e sono utilizzate solo le combinazioni che corrispondono a byte contigui.

1 Collegamento al bus

Consideriamo ora un modulo di memoria da 2^k byte, per esempio $k = 30$ per una memoria di 1 GiB. Possiamo realizzare le funzionalità che ci servono se costruiamo il modulo usando 8 dispositivi di memoria, ciascuno capace di memorizzare $2^k/8 = 2^{k-3}$ byte (nell'esempio precedente, con $k = 30$, ci servono 8 moduli da 128 MiB). I dispositivi devono essere collegati come in Fig. 2. Ciascuna riga di Fig. 1 è memorizzata utilizzando tutti e 8 i dispositivi: il byte della colonna “+0” sarà memorizzato nel dispositivo contrassegnato con 0, il byte “+1” nel dispositivo 1, e così via. Ogni dispositivo memorizza una intera colonna di Fig. 1. Si noti come la presenza dei byte enable semplifichi l'implementazione: ogni byte enable contribuisce a generare il *chip select* del chip che contiene il corrispondente byte.

Il nostro modulo di memoria convive nello spazio di indirizzamento di memoria insieme ad altri dispositivi (altri moduli di memoria, oppure anche interfacce di I/O con registri mappati in memoria). Dobbiamo assegnargli un intervallo di indirizzi e fare in modo che risponda solo alle richieste di lettura e scrittura che ricadono in quell'intervallo. Per farlo conviene scegliere una regione di

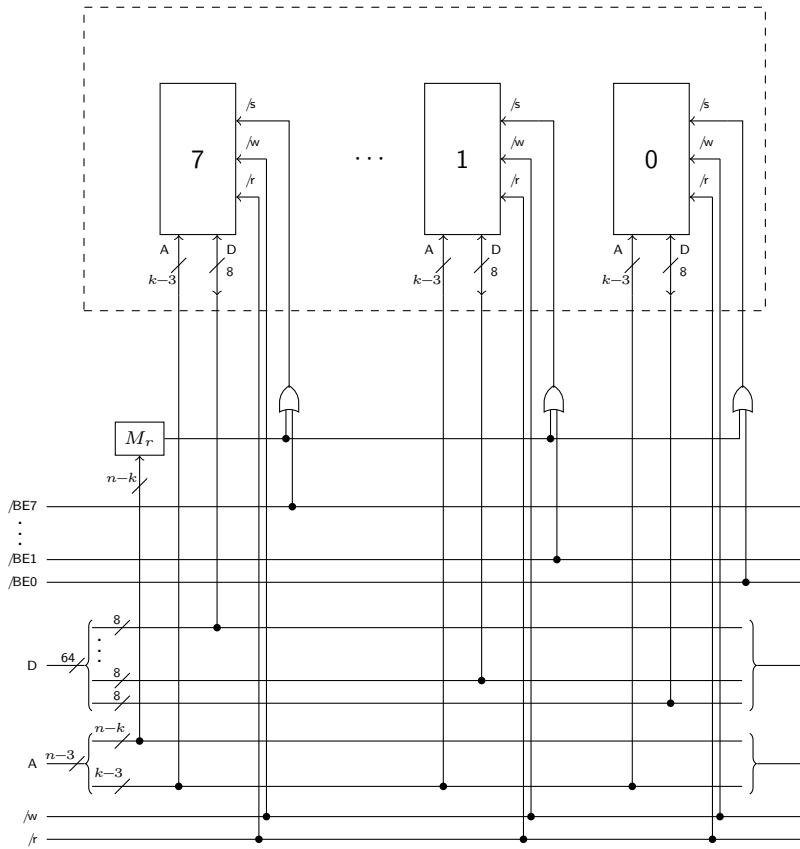


Figura 2: Realizzazione di una memoria organizzata a linee di 64 bit (parole quadruple), ma accessibile anche a byte, parole e doppie parole. Le parti dentro e fuori del rettangolo tratteggiato possono essere realizzate separatamente e poi collegate. La parte interna è una scheda di memoria, la parte esterna è il bus.

2^k byte e fare in modo che la nostra memoria la occupi interamente (facciamo in modo, cioè, che il nostro modulo sia allineato naturalmente all'interno dello spazio di indirizzamento). Sia r il numero della regione grande 2^k che abbiamo scelto. Data una operazione di lettura o scrittura ad un certo numero di riga i , la nostra memoria deve sapere se ignorarla o no in base al fatto che il numero di riga i appartenga o meno alla regione r . Abbiamo già visto come fare a vedere se un indirizzo su n bit appartiene o no ad una regione grande 2^k : è sufficiente guardare gli $n - k$ bit più significativi dell'indirizzo. In questo caso non abbiamo tutto l'indirizzo (di byte) ma solo il numero di riga. Questo non è un problema: ci mancano solo i 3 bit meno significativi dell'indirizzo di byte, e questi sicuramente rientrano nei k che già dovevamo ignorare. Un altro modo per visualizzare l'operazione è di riportare tutto alle righe: una regione di 2^k byte è anche una regione di 2^{k-3} righe. I numeri di regione restano gli stessi che per i byte e per sapere se una riga appartiene o no alla regione scelta è sufficiente ignorare $k - 3$ bit meno significativi del numero di riga e confrontare gli altri con il numero di regione r . I $k - 3$ bit meno significativi, invece, sono l'offset della riga all'interno della regione. Questi possono essere usati per selezionare il byte corretto in ciascun chip di RAM.

Il circuito risultante è quello di Figura 2. La maschera M_r serve ad abilitare o disabilitare complessivamente tutta la scheda di memoria.

La parte interna al rettangolo tratteggiato in Fig. 2 rappresenta la scheda di memoria vera e propria, mentre la parte esterna rappresenta i circuiti del bus a cui la scheda è collegata. Il bus può proseguire a destra e sinistra e avere altre maschere che riconoscono valori di r diversi. Tipicamente avremo una “scheda madre” che prevede un certo numero di slot in cui si possono inserire le schede di memoria. Ogni slot avrà una maschera diversa. Le schede di memoria possono invece essere tutte uguali tra loro ed essere montate in un qualunque slot; più schede possono essere montate contemporaneamente sul bus, ciascuna ovviamente inserita in uno slot diverso.

2 Accessi non allineati

I processori AMD/Intel a 64 bit permettono anche accessi non allineati. Per esempio, è possibile leggere con una sola istruzione una parola quadrupla che inizi ad un indirizzo che non è multiplo di 8:

```
mov 4097, %rax
```

Supponiamo che all'indirizzo 4097 sia memorizzato il numero 1122334455667788 (base 16). I byte che compongono il numero si troveranno nelle posizioni indicate in Figura 3. In questo caso il processore eseguirà due accessi in memoria: uno alla riga numero 511 ($4096/8$) con tutti i byte enable attivi, tranne /BE0; un altro alla riga successiva (512), con tutti i byte enable *non* attivi, tranne /BE0. In questo modo riesce a recuperare tutti i byte che compongono il numero, ma non basta: la prima lettura porterà i byte 22–88 in un registro interno del processore, ma in una posizione che è traslata di 8 bit a sinistra rispetto

numero di riga	+7	+6	+5	+4	+3	+2	+1	+0	indirizzo di riga
...									
510									4080
511	22	33	44	55	66	77	88		4096
512								11	4112
513									4128
...									

Figura 3: Accesso non allineato.

a quella desiderata; la seconda lettura porterà il byte 11 nella posizione meno significativa del registro interno, invece che in quella più significativa. Il processore, automaticamente, provvederà ad eseguire i necessari shift in modo che **rax**, alla fine, contenga il valore corretto.

Si noti come il soggetto di tutte queste azioni è *il processore*: il software contiene solo l'istruzione “`mov 4097, %rax`” e non menziona in alcun modo le due letture e gli shift. Queste sono azioni svolte dal processore per eseguire correttamente l'istruzione richiesta. Possiamo dunque capire che gli accessi non allineati comportano un costo in termini di tempo, e richiedono hardware aggiuntivo nel processore. Alcuni tipi di processori (per es., quelli di ARM) non contengono questo hardware e non ammettono accessi non allineati.

Memoria Cache

G. Lettieri

17 Marzo 2022

1 Introduzione

La memoria centrale è molto più lenta del processore. Possiamo rendercene conto scrivendo un programma che accede ripetutamente agli elementi di un array, misurando quanti cicli di clock sono in media necessari per ogni accesso. La Figura 1 mostra i risultati ottenuti su un processore Intel Core i7-4470 con clock a 3.4 GHz, per varie dimensioni dell'array. Si vede che quando l'array è più grande di 8 MiB, gli accessi in memoria possono richiedere fino a 250 cicli di clock. Si tratta di un tempo enorme. Ricordiamo che il processore è in grado potenzialmente di completare una istruzione per ogni ciclo di clock, ma le istruzioni si trovano in memoria. Se per prelevare una istruzione sono necessari 250 cicli di clock, è del tutto inutile avere un processore velocissimo, in quanto per la stragrande maggioranza del tempo starebbe fermo ad aspettare la prossima istruzione.

In Figura 1, però, notiamo anche che le operazioni di lettura sembrano richiedere molto meno tempo se l'array è sufficientemente piccolo. Per esempio, per array più piccoli di 32 KiB sembrano essere sufficienti 4 cicli per operazione. Questo è l'effetto della memoria *cache*, che ora vogliamo studiare.

2 La memoria Cache

Esistono diverse tecnologie che permettono di costruire una memoria ad accesso casuale. In generale, è possibile costruire memorie grandi ed economiche, ma lente, oppure memorie piccole e veloci, ma costose. Noi vorremmo invece avere memorie grandi, economiche e veloci. Queste non possiamo ottenerle direttamente, ma possiamo avere qualcosa di equivalente usando contemporaneamente i due tipi di memoria e sfruttando alcune caratteristiche dei programmi. Queste caratteristiche prendono il nome di *principi di località*, e sono i seguenti:

- *località spaziale*: se un programma accede ad un certo indirizzo, è molto probabile che in breve tempo accederà ad un indirizzo vicino;
- *località temporale*: se un programma accede ad un certo indirizzo, è molto probabile che in breve tempo vi accederà di nuovo.

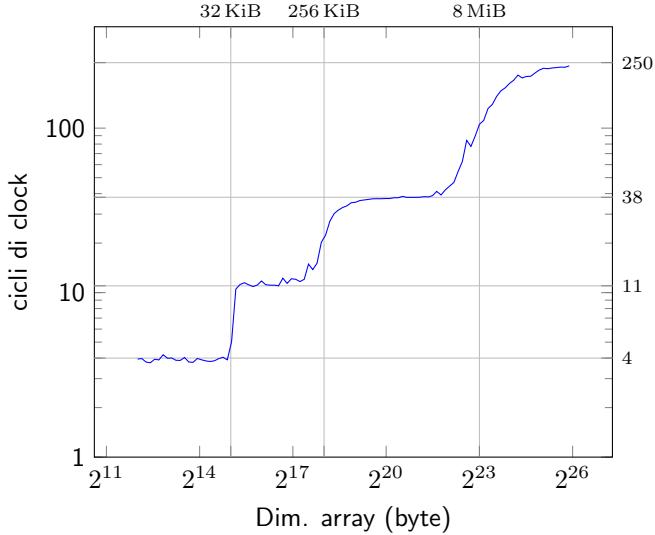


Figura 1: Numero medio di cicli per ogni accesso in memoria per un programma che accede ciclicamente a tutti gli elementi di un array di dimensione variabile.

Questi sono principi puramente statistici che i programmi tipicamente rispettano. Per esempio, le istruzioni sono eseguite per lo più in sequenza, quindi il prelievo di una istruzione ad un certo indirizzo è molto spesso seguito dal prelievo della successiva (quindi ad un indirizzo vicino). La presenza di cicli nel programma comporterà il prelievo ripetuto delle stesse istruzioni. Analoghe considerazioni valgono anche per i dati, in quanto i programmatori li organizzano spesso in strutture dati in cui le variabili usate insieme si trovano vicine, e vi accedono ripetutamente. Quindi, anche se un programma può avere complessivamente bisogno di molta memoria, se lo osserviamo per un intervallo di tempo sufficientemente breve vedremo che si concentra su una parte molto più piccola (magari diversa man mano che l'esecuzione procede). Se riuscissimo a scoprire qual è questa parte e copiarla nella memoria veloce, il nostro programma potrebbe essere eseguito molto più velocemente.

L'idea è di realizzare la memoria centrale con la tecnologia lenta, ma economica, in modo da poterla avere molto grande. Allo stesso tempo aggiungiamo al sistema una memoria cache come illustrato in Figura 2. La cache è composta da un *controllore cache* e dalla memoria cache vera e propria, realizzata con la tecnologia costosa, ma veloce. La memoria cache è dunque molto più piccola della memoria centrale.

Il controllore intercetta tutte le operazioni di lettura e scrittura nello spazio di memoria eseguite dal processore. Per ogni operazione controlla prima se il dato a cui il processore vuole accedere si trova in memoria cache, nel qual caso l'accesso può essere completato velocemente; altrimenti esegue l'operazione in memoria al posto del processore e mantiene una copia del dato in cache (in

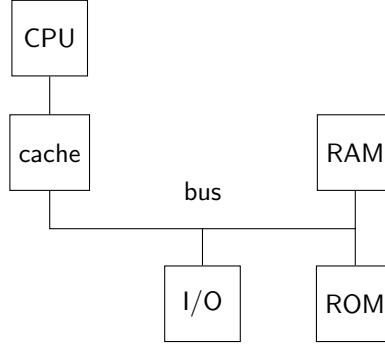


Figura 2: Architettura generale di un calcolatore con memoria cache.

quanto si spera che il processore lo chiederà di nuovo, per il principio di località temporale). All'inizio la memoria cache non conterrà niente, ma a poco a poco si riempirà con le locazioni a cui il programma sta accedendo. Se la memoria cache si riempie, il controllore dovrà decidere quali locazioni tenere e quali eliminare (*rimpiazzare*), sempre cercando di mantenere quelle che è più probabile che vengano richieste in seguito. Si noti che né il controllore cache, né il processore conoscono le locazioni che verranno richieste in futuro: il controllore vede solo le operazioni che il processore genera e il processore vede una sola istruzione per volta. Le decisioni del controllore sono dunque euristiche.

Tipicamente il controllore legge dalla memoria più di quanto il processore ha chiesto, sia per sfruttare il principio di località spaziale, sia per altri motivi che vedremo tra breve. L'unità di trasferimento utilizzata dal controllore cache è detta *cacheline*. Un esempio tipico è di avere cacheline di 64 byte, allineate naturalmente. Per esempio, se il processore esegue una operazione di lettura all'indirizzo 8, il controllore trasferirà dalla memoria tutti i byte che vanno da 0 a 63 (cioè tutta la cacheline che contiene la locazione richiesta dal processore).

Si noti che tutto quanto abbiamo detto si svolge completamente in hardware, nel controllore cache, ed è trasparente al software: i programmi possono essere scritti ignorando che la cache esista e funzioneranno lo stesso. In presenza della cache, però, verranno in genere eseguiti più velocemente.

Anche il processore può ignorare completamente la presenza della cache, nel senso che non sono richieste modifiche al suo funzionamento, purché disponga di un modo per variare la lunghezza delle operazioni di lettura e scrittura in memoria. Questo perché tali operazioni richiederanno tempi molto diversi, a seconda che il dato richiesto si trovi in cache o debba essere prelevato dalla memoria. A tale scopo è sufficiente che il processore possieda un piedino di ingresso tramite il quale il controllore cache può segnalare quando l'operazione si è conclusa, oppure quando deve essere prolungata rispetto ad un tempo di default.

Si noti infine che il meccanismo della cache non ha alcun senso per le operazioni di I/O, in quanto queste operazioni hanno effetti collaterali che non

devono essere cancellati: se un programma vuole leggere il prossimo carattere battuto sulla tastiera è necessario interpellare la tastiera. Non avrebbe alcun senso re-inviare al processore sempre lo stesso carattere conservato in cache. Il controllore cache, dunque, farà passare inalterate tutte le operazioni di lettura e scrittura nello spazio di I/O. Questo però non basta: si ricordi che le interfacce di I/O possono anche avere i loro registri mappati nello spazio di memoria. Per disabilitare la cache anche durante gli accessi a queste interfacce è necessario dunque operare una discriminazione anche in base all'indirizzo usato dal processore, ma per vedere come fare dovremo aspettare di aver introdotto la paginazione.

3 Cache ad indirizzamento diretto

La memoria cache è, dal punto di vista funzionale, una normale memoria ad accesso casuale: le operazioni possibili sono sempre quelle di lettura e scrittura, eseguite una per volta, ognuna ad un certo *indirizzo di cache*.

Il controllore cache intercetta le operazioni di lettura e scrittura del processore. Ognuna di queste è relativa ad un certo indirizzo di memoria. Si ricordi che il processore genera l'indirizzo di una certa parola quadrupla allineata naturalmente, che abbiamo chiamato *numero di riga*, usando poi i fili di byte enable per selezionare i byte all'interno della riga selezionata. La memoria cache sarà organizzata nello stesso modo, così che le operazioni di lettura e scrittura del processore si mappino facilmente su quelle della cache.

Il controllore, però, lavora con unità più grandi, dette cacheline, che per esempio possono essere di 8 parole quadruple (64 byte), allineate naturalmente. Il controllore può dunque scomporre il numero di riga generato dal processore in una parte meno significativa detta *offset* (di 3 bit se le cacheline sono di 8 parole quadruple) e nel rimanente numero di cacheline. Poiché il controllore trasferisce sempre intere cacheline, il numero di cacheline è sufficiente per determinare se la locazione richiesta dal processore è in cache oppure no.

Dato il numero di cacheline, il controllore cache deve essere in grado di sapere se la corrispondente cacheline di memoria è stata precedentemente copiata in cache e, in caso affermativo, a quale indirizzo di cache. Un modo per realizzare questo meccanismo è di avere una funzione hash da numeri di cacheline a indirizzi di cache. Nelle cache a *indirizzamento diretto* questa funzione è estremamente semplice (e dunque veloce): l'indirizzo di cache (detto *indice*) è dato dai bit meno significativi del numero di cacheline. In particolare, se la cache è grande 2^a cacheline, l'indice sarà di a bit. Si noti che tutti i numeri di cacheline che distano tra loro 2^a cacheline avranno lo stesso indice, e dunque vorrebbero essere memorizzate nella stessa posizione della cache (si dice che c'è un conflitto tra le due cacheline). Il controllore deve sapere quale tra le tante cacheline che potrebbero trovarsi ad un certo indirizzo di cache è quella effettivamente caricata. Queste diverse cacheline si distinguono per la parte del numero di cacheline che non è utilizzata nell'indice e che è detta *etichetta*. Il controllore può utilizzare una memoria aggiuntiva, detta memoria delle etichette (o *tag*), in cui

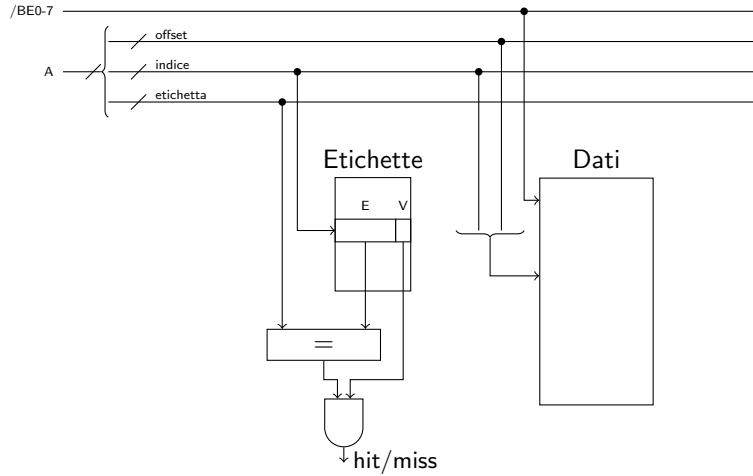


Figura 3: Cache a indirizzamento diretto.

memorizzare, per ogni indice, l'etichetta della cacheline caricata a quell'indice. Il controllore deve anche sapere a quali indici non è stata caricata ancora alcuna cacheline. Poiché le memorie contengono sempre qualcosa e qualunque sequenza di bit potrebbe essere una valida etichetta, è necessario memorizzare un bit in più per ogni indice, detto *bit di validità*, che valga 1 se e solo se l'etichetta (e dunque la cacheline) contenuta a quell'indice è significativa. All'inizio tutti i bit di validità saranno a 0; passeranno a 1 man mano che il controllore cache caricherà cacheline in risposta agli accessi in memoria del processore. Lo schema è dunque quello di Figura 3.

Per ogni operazione di lettura in memoria da parte del processore, le operazioni svolte dal controllore cache saranno le seguenti:

- se la cacheline è presente (*read hit*), completa la lettura con i dati presenti in cache; per leggere dalla memoria cache dati la parola quadrupla richiesta dal processore è sufficiente usare come indirizzo l'indice e l'offset;
- se la cacheline non è presente (*read miss*), il controllore deve leggere la cacheline della memoria, scriverla nella cache dati a partire dalla posizione dettata dall'indice, e aggiornare l'etichetta; a questo punto i dati sono in cache e si procede come nel caso precedente.

Si noti che, in caso di miss, una eventuale cacheline che si trovava in cache allo stesso indice di quella appena caricata verrà rimpiazzata. Questo è il difetto principale delle cache ad indirizzamento diretto: due cacheline che hanno lo stesso indice non possono essere mantenute contemporaneamente in cache, anche se il resto della cache è vuota.

Per le operazioni di scrittura abbiamo diverse opzioni.

- se la cacheline è assente (*write miss*), il controllore può completare la scrittura in memoria senza caricare la cacheline in cache (*write no-allocate*) oppure caricare la cacheline in cache (*write allocate*) e proseguire come in una *write hit*;
- se la cacheline è presente (*write hit*), il processore può aggiornare solo la cache (*write back*) o sia la cache che la memoria (*write through*).

Si noti che nel caso di *write back* la cacheline dovrà essere riscritta in memoria prima di essere rimpiazzata da un'altra cacheline, perché altrimenti le scritture eseguite dal programma andrebbero perse. La tecnica è comunque conveniente se il programma riesce ad eseguire tante scritture sulla stessa cacheline prima che questa debba essere scritta, in quanto si riduce il numero di scritture in memoria. Per evitare scritture inutili, inoltre, il controllore cache può associare ad ogni cacheline un bit *dirty* e porlo a 1 quando il processore ha eseguito almeno una scrittura in quella linea. Le cacheline che hanno il bit dirty a zero non hanno bisogno di essere riscritte in memoria. Le cache moderne sono tipicamente write-allocate e write-back.

La memoria delle etichette è un costo aggiuntivo ed è bene che sia sufficientemente piccola. Notiamo che abbiamo bisogno di una etichetta per ogni cacheline della memoria dati. Se usiamo cacheline più grandi ne riduciamo il numero, e quindi riduciamo anche la dimensione della memoria delle etichette. Questo è il motivo principale per cui la cacheline contiene più byte di quanti ne può richiedere il processore in una singola operazione. D'altro canto, la cacheline non deve essere nemmeno troppo grande, altrimenti leggere o scrivere una cacheline dalla memoria richiederebbe troppo tempo. La dimensione di 64 byte è un buon compromesso tra queste due esigenze contrastanti.

In presenza di cache non è più il processore ad accedere alla memoria centrale, ma sempre il controllore cache. Dal momento che questo trasferisce sempre cacheline intere, si può ottimizzare questo tipo di trasferimento. Per esempio, con una cacheline di 64 byte e bus di 64 bit, sono necessarie 8 operazioni di lettura per trasferire una cacheline, ciascuna delle quali richiede il trasferimento dell'indirizzo. Ma dal momento che gli indirizzi sono consecutivi, possiamo ottimizzare l'operazione facendo in modo che la memoria centrale risponda sempre con 8 parole quadruple in sequenza ad ogni operazione di lettura (e dualmente per la scrittura). In questo modo possiamo trasferire l'indirizzo una sola volta per tutta la cacheline.

4 Cache associative ad insiemi

Le cache associative ad insiemi sono più costose di quelle ad indirizzamento diretto, ma permettono di alleviare il problema dei conflitti. L'idea è di permettere la memorizzazione in cache di più di una cacheline per ogni indice. Una cache associativa ad insiemi che permette di memorizzare n cacheline per ogni indice è detta a n vie. In Figura 4 è mostrato lo schema di una cache associativa a 2 vie. Al suo interno ci sono due repliche di una cache ad indirizzamento

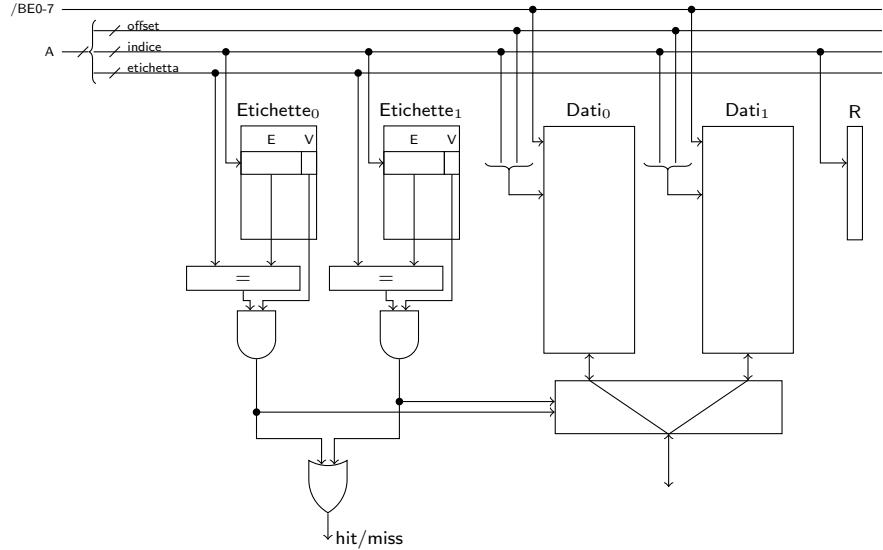


Figura 4: Cache associativa a insiemi (a 2 vie).

diretto, collegate in parallelo. Ogni cacheline ha sempre un indice, ma questo ora individua due possibili locazioni: una in $Dati_0$ e una in $Dati_1$. Ogni cacheline può essere memorizzata indifferentemente nell'una o nell'altra. Ogni indice, dunque, individua un *insieme* di possibili locazioni (di cache) per la cacheline.

Si avrà un hit se una delle vie dell'insieme contiene la cacheline cercata, da cui la porta OR che riceve le uscite delle due porte AND. La cacheline a cui accedere dipenderà ovviamente da quale via ha prodotto l'hit. (Si noti che non è possibile che entrambe le vie producano hit, in quanto una cacheline viene caricata solo se non è già presente, e all'inizio la cache è vuota.) In Figura 4 questo è indicato sommariamente dal circuito che si trova sotto le due memorie Dati.

In caso di miss, il controllore può scegliere quale delle due cacheline rimpiazzare. La politica più usata in questo caso è la LRU (Least Recently Used): si rimpiazza la cacheline che non è acceduta da più tempo. Questa politica è quella che si comporta meglio nella maggior parte dei casi, anche se non è ottima in assoluto (in alcuni casi è anzi la peggiore). Purtroppo la politica matematicamente ottima richiede la conoscenza di tutti gli accessi futuri e non è realizzabile in pratica.

La memoria R in Figura 4 contiene le informazioni necessarie ad implementare la politica LRU. Con solo due vie è sufficiente ricordare, per ogni indice, la via riferita dall'ultimo accesso: la via da rimpiazzare in caso di miss sarà evidentemente l'altra. Con più di due vie sarebbe però necessario ricordare l'ordine degli ultimi accessi di ogni via, e aggiornarlo ad ogni accesso, operazione che potrebbe essere troppo costosa. In pratica conviene utilizzare politiche

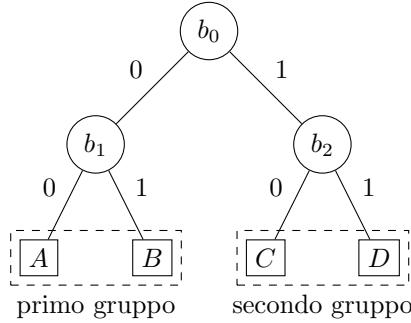


Figura 5: Scelta della via da rimpiazzare nell'algoritmo pseudo-LRU.

approssimate, se non addirittura effettuare un rimpiazzamento casuale.

Per le cache a 4 vie esiste un interessante algoritmo che approssima LRU ed è molto semplice da realizzare. L'algoritmo si chiama Pseudo-LRU e richiede 3 bit per ogni insieme, chiamiamoli b_0 , b_1 e b_2 . Chiamiamo A , B , C e D le quattro vie. L'algoritmo raggruppa le vie a due due: A e B da una parte e C e D dall'altra. Il bit b_0 dirige la scelta sul primo gruppo o sul secondo gruppo:

- se $b_0 = 0$ la via da rimpiazzare è una tra A e B . La scelta tra le due è decisa dal valore di b_1 , ignorando b_2 :
 - se $b_1 = 0$ si sceglie A ;
 - se $b_1 = 1$ si sceglie B ;
- se $b_0 = 1$, invece, la scelta è tra C e D (secondo gruppo) ed è decisa dal valore di b_2 , ignorando b_1 :
 - se $b_2 = 0$ si sceglie C ;
 - se $b_2 = 1$ si sceglie D ;

Si veda anche la Figura 5. Ad ogni accesso vanno aggiornati due bit, lasciando inalterato il terzo, in base alla via interessata dall'accesso. L'idea è che la via appena acceduta deve diventare l'ultima ad essere rimpiazzata, così come accadrebbe nell'algoritmo LRU:

- accesso a A : $b_0 \rightarrow 1$ e $b_1 \rightarrow 1$;
- accesso a B : $b_0 \rightarrow 1$ e $b_1 \rightarrow 0$;
- accesso a C : $b_0 \rightarrow 0$ e $b_2 \rightarrow 1$;
- accesso a D : $b_0 \rightarrow 0$ e $b_2 \rightarrow 0$.

Il fatto di lasciare sempre inalterato uno tra b_1 o b_2 permette di ricordare l'ordine relativo tra le vie del gruppo non interessato dall'accesso. D'altro canto, la via che si trova nello stesso gruppo di quella interessata dall'accesso può ora trovarsi

più in altro in coda (grazie al valore scritto in b_0) rispetto a dove sarebbe stata nel vero algoritmo LRU (da qui l'approssimazione).

Si noti inoltre che, se la memoria R è organizzata in tre banchi indipendenti da un bit ciascuno, ogni aggiornamento può essere implementato con una operazione di scrittura in due dei tre banchi, senza la necessità di dover prima eseguire una operazione di lettura.

Per cache con più di 4 vie si utilizzano in genere algoritmi Pseudo-LRU modificati, in cui per esempio i livelli più bassi dell'albero mancano e sono sostituiti da scelte casuali.

In Fig. 1 vediamo che il sistema testato contiene in realtà più cache in cascata: una prima cache di 32 KiB, una seconda cache (a cui accede quando non trova una cacheline nella prima) grande 256 KiB e una terza cache (a cui accede quando non trova una cacheline nelle prime due) grande 8 MiB. Le cache più grandi sono via via più lente, con accessi di circa 10 e 40 cicli clock, rispettivamente. Questo può essere confermato con vari strumenti (per es., in Linux su Intel/AMD, con il comando `lscpu --cache`) e in questo modo è possibile anche sapere che le prime due cache sono in realtà associative a 8 vie, mentre la terza cache è associativa a 16 vie. Il salto netto da 4 a 10 cicli non appena l'array supera la dimensione della prima cache è tipico dell'algoritmo LRU realizzato esattamente. Invece, gli andamenti più smussati in corrispondenza dei passaggi alla seconda e alla terza cache indicano che queste cache usano una versione approssimata di LRU, probabilmente un mix tra Pseudo-LRU e rimpiazzamento casuale.

Accesso a basso livello

G. Lettieri

17 Marzo 2022

1 Ambiente protetto

Nella nostra comune esperienza di programmatore, 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 assemblatore 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 mostrarceli 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, assemblatore 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++¹.

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.

¹Si 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 proseguiamo 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 esadecimale. 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 RAM² 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.

²Per 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.

Periferiche

G. Lettieri

18 Marzo 2022

Studiamo ora le periferiche essenziali di un PC. Faremo riferimento alle vecchie periferiche del PC AT, sia perché sono più semplici da programmare dei loro equivalenti moderni, sia perché molti PC moderni continuano a supportarle, per compatibilità software. In particolare, sono supportate dalla macchina virtuale QEMU che usiamo per sviluppare tutti gli esempi. In ogni caso i concetti base sono rimasti sostanzialmente gli stessi nel tempo.

Per ogni periferica daremo una breve descrizione del suo funzionamento interno, quindi descriveremo l'interfaccia visibile al programmatore, almeno quanto basta per poter programmare dei semplici esempi. Il codice degli esempi può essere scaricato da

<https://calcolatori.iet.unipi.it/resources/esempioIO-7.2.tar.gz>

Tutti gli esempi fanno uso della libreria `libce`, scaricabile dallo stesso sito.

1 Tastiera

Lo scopo della tastiera è di rilevare i tasti premuti e rilasciati auto comunicarlo al PC. A (quasi) ogni tasto fisico sono associati due codici numerici: il *make code*, che indica che il tasto è stato premuto, e il *break code*, che indica che il tasto è stato rilasciato. Questi codici sono associati al tasto e non alla sua funzione: per esempio, i due tasti shift hanno codici diversi. È il software che assegna (liberamente) un significato ai tasti.

Per svolgere il proprio compito, le tastiere contengono tipicamente una matrice di collegamenti elettrici. Nelle tastiere più comuni (ed economiche) questa è realizzata con tre fogli di plastica sovrapposti, chiamiamoli 1, 2 e 3. Sul foglio 1 sono tracciati dei collegamenti verticali e sul foglio 3 dei collegamenti orizzontali. Le tracce si incrociano in corrispondenza di ciascun tasto, ma sono tenute separate dal foglio 2, che si trova in mezzo agli altri due. Il foglio 2, però, contiene un buco in corrispondenza di ogni tasto (e dunque di ogni incrocio). Ogni volta che si preme un tasto si mette in collegamento, attraverso il buco, una traccia verticale del foglio 1 con una traccia orizzontale del foglio 3, semplicemente esercitando una pressione sul foglio superiore tramite un bastoncino di gomma che si trova sotto il tasto.

La matrice è monitorata da un *microcontrollore* che si trova a bordo della tastiera (originariamente un Intel 8042). Un microcontrollore è un piccolo

computer, con una sua CPU, una sua RAM, una ROM e delle porte di I/O, interamente contenuto in un unico chip. Le porte di I/O del microcontrollore sono direttamente collegate a dei piedini del chip. Nella tastiera, i piedini di I/O del microcontrollore sono collegati alle tracce verticali e orizzontali della matrice di cui sopra. Il microcontrollore è programmato in modo da inviare un segnale su una colonna alla volta e leggere lo stato di tutte le righe della matrice. Se un tasto di quella colonna è premuto, il segnale ritornerà sulla corrispondente riga: in questo modo il controllore può rilevare la pressione di un tasto¹. Il microcontrollore passa poi alla colonna successiva, e così via. La scansione viene ripetuta migliaia di volte al secondo, cosa che è sufficiente a non perdere le battute anche del più veloce dattilografo. Per rilevare quando un tasto è stato rilasciato, il microcontrollore ricorda lo stato di ogni tasto nella propria RAM. Quando un tasto cambia stato, il microcontrollore trasmette un appropriato codice di scansione al PC.

L'informazione di "tasto rilasciato" (break code) serve al software per gestire le combinazioni di tasti (se ricevo il make code di shift seguito dal make code del tasto 'a', prima di aver visto il break code del tasto shift, vuol dire che l'utente sta cercando di scrivere 'A'). Le tastiere dei PC, invece, gestiscono autonomamente i tasti *typematic*: se il controllore della tastiera si accorge che uno dei tasti typematic è sempre premuto per tutto un certo intervallo di tempo (configurabile), inizia a inviare il make code di quel tasto ripetutamente, con un certo ritmo (anch'esso configurabile)².

La comunicazione tra tastiera e PC può avvenire in vari modi. Nel seguito facciamo riferimento alle tastiere di tipo PS/2, la cui interfaccia di programmazione rispecchia ancora quella del PC AT. In questo caso la comunicazione tra PC e tastiera è su un cavo seriale, e all'altro capo del cavo, sulla scheda madre del PC, si trova un altro microcontrollore (originariamente anch'esso un Intel 8042), che riceve i codici, li converte nei corrispondenti make code o break code (la conversione avviene, al solito, per motivi di compatibilità software), e li rende disponibili in un registro mappato nello spazio di I/O del bus, da cui il software può poi leggerli. I make code e break code sono su 8 bit (per quasi tutti i tasti). Inoltre, il break code differisce dal corrispondente make code solo per il bit più significativo (1 per i break code e 0 per i make code).

1.1 Interfaccia per il programmatore

Il programmatore interagisce esclusivamente con il microcontrollore a bordo del PC, tramite una interfaccia che espone quattro registri: un registro di lettura, RBR, un registro di stato STR, un registro di scrittura, TBR, e un registro di controllo, CMR, tutti da 8 bit. Questi registri occupano solo due indirizzi dello spazio di I/O, in modo da risparmiare piedini di indirizzo nell'interfaccia: RBR

¹Per semplicità evitiamo di discutere i problemi che si verificano quando più di un tasto per colonna o riga sono contemporaneamente premuti (*ghosting* e *jamming*), o dei limiti al cosiddetto *roll-over*, cioè del massimo numero di tasti contemporaneamente premuti che la tastiera può correttamente rilevare e/o riportare.

²Non tutti i tasti sono typematic. Per esempio, i tasti shift, ctrl, alt non lo sono.

e TBR sono entrambi all'indirizzo 0x60, mentre STR e CMR sono entrambi all'indirizzo 0x64. Questo è possibile perché RBR e STR sono di sola lettura, mentre TBR e CMR sono di sola scrittura.

I bit 0 e 1 del registro STR fungono da flag “busy” per i registri RBR e TBR, rispettivamente. In particolare, il programmatore deve assicurarsi di leggere RBR solo se il bit 0 di STR è 1, per essere sicuro di star leggendo un nuovo codice.

Tramite il microcontrollore a bordo del PC è possibile anche *inviare* comandi alla tastiera, per esempio per accendere o spegnere i led, o configurare i parametri del typematic. Questo è uno degli scopi dei registri CMR e TBR (vedremo un altro utilizzo di questi registri quando parleremo delle interruzioni).

La cartella `esempiIO` contiene due programmi che utilizzano l'interfaccia della tastiera: `tastiera-1` e `tastiera-2`.

L'esempio `tastiera-1` legge ciclicamente un nuovo codice e lo stampa sul video in binario. L'esempio fa uso di alcuni tipi e funzioni definite in `libce`:

1. `ioaddr` è un **typedef** per **unsigned short** (16 bit) ed è utilizzato per definire gli indirizzi nello spazio di I/O;
2. `inputb()` serve a leggere un byte dallo spazio di I/O, all'indirizzo passato come argomento;
3. `char_write()` serve a mostrare un carattere sullo schermo.

La funzione `inputb()` è definita in assembly (nel file `64/inputb.s` nei sorgenti di `libce`) in modo da poter utilizzare l'istruzione **inb**, che è sconosciuta al compilatore C++. La funzione `char_write()` la vedremo quando parleremo del video. La funzione `get_code()` legge ciclicamente STS fino a quando non trova il bit 0 settato a 1, quindi legge il nuovo codice da RBR. Si noti che una funzione sostanzialmente identica è definita anche in `libce`, e negli esempi successivi useremo direttamente quella offerta dalla libreria.

Il programma `tastiera-2` vuole mostrare un semplice esempio di operazioni tipicamente svolte dal software, come convertire i codici letti dalla tastiera in codici ASCII, tenendo conto dello stato di tasti modificatori come lo shift, e farne l'echo sul video. In particolare, la funzione `char_read()` legge un nuovo codice dalla tastiera e tiene traccia dello stato dello shift sinistro nella variabile globale `shift`. Il **while** serve ad ignorare i codici dello shift, che abbiamo già gestito, e i break code degli altri tasti, che non ci interessano (tutti i break code sono maggiori di 0x80). La funzione `conv()` provvede poi ad associare un codice ASCII al make code ricevuto, usando la tabella `tabmin` o `tabmai`, in base allo stato corrente del tasto shift. La libreria `libce` contiene versioni di `conv()` e `char_read()` identiche a quelle mostrate in questo esempio; contiene anche le tabelle `tab`, `tabmin` e `tabmai`, ma con qualche codice in più.

2 Video

L’interfaccia video ruota attorno all’idea della *memoria video*. Si tratta di una memoria destinata a contenere una descrizione di ciò che deve apparire sullo schermo. Alla memoria video accedono concorrentemente sia il software (normalmente in scrittura), sia un *controllore video*, che la legge completamente (per esempio 60 o 90 volte al secondo) e ne interpreta il contenuto per generare i segnali per il monitor (analogici nei vecchi monitor VGA, digitali nei monitor moderni). Tranne che nelle schede video più economiche, la memoria video si trova a bordo della stessa scheda video che contiene anche il controllore ed è mappata nello spazio di indirizzamento di memoria del bus. Questo vuol dire che, per il programmatore, accedere alla memoria video è sostanzialmente come accedere ad una struttura dati. A livello assembly è possibile usare tutte le istruzioni che ammettono un operando in memoria (tipicamente `mov`).

Oltre a scrivere e leggere dalla memoria video, il software può dialogare anche con il controllore stesso. Quest’ultimo ha una interfaccia con vari registri che, nella scheda che vedremo, sono accessibili nello spazio di I/O. Scrivendo opportunamente in questi registri è possibile, in particolare, selezionare la *modalità video*, che stabilisce come deve essere interpretato il contenuto della memoria video. Esistono due modalità principali, con varie sotto-modalità:

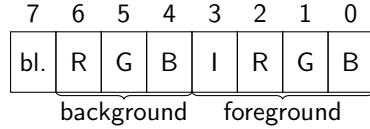
- *Modalità testo*: in questo caso la memoria video contiene i codici (per es. ASCII) dei caratteri che devono essere visualizzati sullo schermo, più eventualmente altri attributi come il colore dei singoli caratteri;
- *Modalità grafica*: in questo caso la memoria video serve a specificare il colore di ciascun pixel dello schermo.

Nel caso della modalità testo, il controllore video ha anche bisogno di un *font* che gli dica come disegnare sullo schermo i vari caratteri. Normalmente uno o più font sono contenuti in una ROM a bordo della scheda video. Le varie sottomodalità della modalità testo si distinguono per il numero delle righe e colonne in cui è suddiviso lo schermo. Le varie sotto-modalità della modalità grafica si distinguono invece per la risoluzione (normalmente espressa in pixel orizzontali per pixel verticali) e per il numero di colori possibili per ciascun pixel. Lo standard SVGA (Super Video Graphics Array) definisce i dettagli di varie modalità, sia testo che grafiche, che le schede che supportano lo standard devono offrire. Nel seguito esaminiamo la modalità testo 80×25 (80 righe per 25 colonne), che è quella che tradizionalmente le schede video per PC offrono di default, e le modalità grafiche di tipo *framebuffer*, che sono le più semplici da usare.

Il controllore contiene un gran numero di registri interni, ma, per risparmiare piedini, espone al programmatore due soli registri: IND e DAT. Tramite questi registri il programmatore può accedere *indirettamente* a qualunque registro interno. Ogni registro interno è identificato da un indice e, per accedervi, il programmatore deve prima scrivere l’indice desiderato nel registro IND; a quel punto il registro DAT diventa una “finestra” sul registro interno selezionato: scrivendo e leggendo da DAT si legge e scrive nel registro interno.

2.1 Modalità testo

La scheda video emulata da QEMU si trova all'avvio in modalità testo 80×25 . In questa modalità lo schermo è idealmente diviso in una matrice di 80 per 25 posizioni, ciascuna delle quali può visualizzare un carattere. La memoria video è organizzata in una corrispondente matrice di 80×25 elementi, memorizzata per righe. Il primo elemento della matrice si trova all'indirizzo 0xb8000 e corrispondente alla posizione in alto a sinistra dello schermo. Ciascun elemento della matrice è grande due byte: il byte meno significativo contiene il codice del carattere da visualizzare, mentre il byte più significativo contiene l'*attributo colore*. Per quanto riguarda il codice del carattere, normalmente le schede video dei PC accettano un qualunque codice ASCII (7 bit) eventualmente esteso a 8 bit per includere altri caratteri come, in Italiano, le lettere accentate. L'attributo colore, invece, ha il seguente formato:



dove i bit contrassegnati con *foreground* selezionano il colore di primo piano su 4 bit (codificato come Intensity, Red, Green, Blue) e i bit contrassegnati con *background* selezionano il colore di sfondo (codificato come Red, Green, Blue). Il bit 7 (*blinking*), se attivo, rende il carattere lampeggiante, ma questa funzionalità non è emulata da QEMU. Quindi, per esempio, il byte 01001011 (4B in esadecimale) seleziona un colore azzurro chiaro (I+G+B) su sfondo rosso (R).

La cartella `esempioIO` contiene due esempi sul video in modalità testo: `video-testo-1` e `video-testo-2`.

L'esempio `video-testo-1` si limita a inizializzare tutta la memoria video con lo stesso carattere e lo stesso attributo colore. Il tipo `natw`, utilizzato nel codice, è definito in `libce` come un **typedef** per **unsigned short**, ed è dunque grande due byte, quanto ciascun elemento della memoria video. Per accedere alla memoria video direttamente dal C++ è sufficiente dichiarare un puntatore a `natw` (chiamato `video` nel codice) e inizializzarlo con il valore numerico, noto, del primo indirizzo della memoria video (0xb8000, come detto). Per farlo accettare dal compilatore è però necessario un cast, visto che stiamo cercando di assegnare un intero ad un puntatore. Fatto questo, `video` può essere usato come un normale array di `natw`. Gli elementi da 0 a 79 corrispondono alla prima riga di caratteri, quelli da 80 a 159 alla seconda, e così via. Ogni `natw` deve contenere l'attributo colore nel byte più significativo e, nel byte meno significativo, il codice ASCII del carattere da visualizzare. Dopo aver inizializzato la memoria video, il programma attende che l'utente prema `ESC`, chiamando ripetutamente la funzione `char_read()` definita in `libce`, analoga a quella vista in `tastiera-2`.

L'esempio `video-testo-2` mostra come realizzare la funzione usata negli esempi precedenti per mostrare un carattere sul video, `char_write()`. La fun-

zione tiene traccia, nelle variabili `x` e `y`, della posizione in cui dovrà apparire il prossimo carattere, in modo da emulare un “terminale a stampante”, che stampa i caratteri uno a fianco all’altro. La funzione si preoccupa anche di andare automaticamente alla riga successiva quando si raggiunge l’ultima colonna, e di mostrare lo “scroll” verso l’altro del contenuto del monitor quando si raggiunge l’ultima riga. Lo scroll è realizzato copiando ogni riga dello memoria video nella sua riga superiore, quindi riempendo di spazi la riga in fondo. La funzione gestisce anche i caratteri di a-capo, che si limitano a spostare la posizione del prossimo carattere, senza stampare niente. Infine, la funzione sfrutta la possibilità, offerta dal controllore video, di mostrare un *cursore* nella posizione del prossimo carattere. Per farlo è sufficiente scrivere la posizione desiderata in due registri interni del controllore, cosa che è svolta dalla funzione `cursore()`.

2.2 Modalità grafica

Per attivare la modalità grafica, visto che la scheda che stiamo considerando parte in modalità testo, è necessario impostare un gran numero di registri interni. Questa operazione è molto complessa, tanto che molte schede contengono una ROM in cui sono presenti delle funzioni che svolgono questo compito. Noi “bareremo” un po’, in quanto la scheda emulata da QEMU, a sua volta presa dall’emulatore Bochs, può essere configurata molto più facilmente, sostanzialmente scrivendo la risoluzione desiderata in alcuni registri. Queste operazioni sono svolte dalla funzione `bochsvga_config()`, definita in `libce`. La funzione riceve il numero di colonne e righe di pixel desiderato, mentre il numero di colori è fissato a 256. La funzione restituisce l’indirizzo del *framebuffer*, che è l’analogo della memoria video in modalità testo, solo che questa volta l’array contiene una posizione per ciascun pixel dello schermo (nella risoluzione selezionata). Ciascun elemento dell’array è grande un byte, in modo da poter contenere uno qualunque dei 256 colori possibili. Il tipo del valore restituito dalla funzione è direttamente un puntatore a nat`b` (`unsigned char`), in modo che lo si possa usare senza dover fare un cast.

A questo punto, per disegnare qualcosa sullo schermo, è sufficiente colorare opportunamente i pixel desiderati, semplicemente scrivendo il codice del colore nella posizione corrispondente del framebuffer. Si vedano i programmi `svga-1`, `svga-2` e `sgva-3` per tre semplici esempi: uno che colora tutto lo schermo di rosso, uno che disegna un bordo e degli assi, e uno che aggiunge una parabola e una retta.

3 Timer (conteggio)

Il PC AT conteneva una interfaccia di *conteggio* usata per vari scopi. Molti PC moderni continuano a emulare questa interfaccia, e la troviamo anche in QEMU. Queste interfacce decrementano un contatore interno ogni volta che si verifica un evento, segnalato da un impulso su un loro piedino di ingresso. In particolare, se questo piedino è collegato ad un generatore di clock, l’interfaccia

permette di misurare il passaggio del tempo. Il valore iniziale del contatore può essere impostato via software, scrivendo in un apposito registro dell’interfaccia. La scrittura, normalmente, funge anche da *trigger* che avvia il conteggio. Quando il contatore arriva a zero, l’interfaccia genera un segnale su un piedino di uscita, con varie possibili forme d’onda. Queste interfacce possono funzionare a *ciclo singolo* o a *ciclo continuo*. Nel secondo caso l’interfaccia, a fine conteggio, ricarica automaticamente il contatore e fa ripartire il conteggio.

Il chip che si trovava a bordo del PC AT era l’Intel 8254, che conteneva tre interfacce di conteggio indipendenti, chiamate 0, 1 e 2, ciascuna con un contatore a 16 bit. Tutte e tre le interfacce erano collegate a un generatore di clock a 1,190 MHz. Ciascuna interfaccia era indipendentemente configurabile per selezionare, per esempio, il ciclo singolo o continuo, oppure il tipo di forma d’onda da generare (impulso, onda quadra, ...). L’interfaccia 0 era collegata, come vedremo, al controllore delle interruzioni. L’interfaccia 1 era utilizzata per il refresh della memoria dinamica (e non è più emulata nei PC moderni). Infine, l’interfaccia 2 era collegata all’unico dispositivo audio fornito, per default, dal PC AT: un “cicalino” in grado di produrre dei *beep*. Si tratta in pratica di un piccolo altoparlante, una membrana che può essere messa in vibrazione agendo su un elettromagnete.

3.1 Interfaccia per il programmatore

Il chip 8254 contiene 13 registri da 8 bit, 4 per ogni interfaccia di conteggio e uno a comune. Per ogni interfaccia, due registri sono di sola scrittura e permettono di impostare la parte alta e bassa del contatore (CTR_LSB e CTR_MSB); due registri sono di sola lettura e permettono di leggere lo stato corrente del contatore (STR_LSB e STR_MSB). Il registro comune (CWR) serve principalmente a configurare le interfacce. Il chip, però, possiede solo 2 piedini di indirizzo, e dunque occupa solo 4 indirizzi dello spazio di I/O, in particolare gli indirizzi 0x40–0x43. L’indirizzo 0x43 permette di accedere al registro comune, CWR. L’indirizzo 0x40 è utilizzato per tutti e 4 i registri dell’interfaccia 0, l’indirizzo 0x41 per i 4 dell’interfaccia 1 (non più utilizzato, come abbiamo detto), e l’indirizzo 0x42 per i 4 dell’interfaccia 2. Scrivendo all’indirizzo 0x40 si accede alternativamente al registro CTR_LSB e CTR_MSB del contatore 0, mentre leggendo si accede alternativamente ai registri STR_LSB e STR_MSB, e così per gli altri due contatori. In pratica, per impostare il valore completo di un contatore, è necessario eseguire due scritture consecutive allo stesso indirizzo di I/O.

Dal momento che non conosciamo ancora le interruzioni, vediamo per il momento un esempio che usa il cicalino, e dunque il contatore 2. L’idea è che questo contatore deve essere programmato in modo da generare un’onda quadra a ciclo continuo, con un certo periodo. Questa onda quadra pilota (tramite un amplificatore e un filtro passa basso) direttamente l’elettromagnete dell’altoparlante, e dunque fa vibrare la membrana con la stessa frequenza, producendo una nota udibile. Il PC contiene anche un registro SPR, all’indirizzo di I/O 0x61, che serve a controllare più finemente l’ingresso dell’altoparlante.

In particolare, il bit 1 del registro SPR è in AND con l'uscita del contatore 2, e permette dunque di silenziare l'altoparlante se posto a 0; il bit 0 di SPR, invece, è in ingresso al contatore 2 e, se posto a 0, mette in pausa il conteggio.

La cartella `esempioIO` contiene il programma `timer`, che usa questo contatore per generare un sibilo a 1000 Hz. La funzione `avvia_timer()` configura e inizializza il contatore 2. La scrittura del valore 0xB6 in CWR imposta il contatore 2 per generare un'onda quadra a ciclo continuo. Le due successive scritture (si noti che `iCTR2_LSB` e `iCTR2_MSB` sono in realtà lo stesso indirizzo) impostano il valore del contatore a 1190, in modo che il contatore arrivi a zero 1000 volte al secondo. La funzione `main` provvede anche a scrivere 3 (11 in binario) nel registro SPR, in modo da *non* mettere in pausa il conteggio e permettere al segnale del contatore di raggiungere l'altoparlante. La funzione `pause()`, definita in `libc.c`, scrive un messaggio sul video e aspetta che l'utente prema ESC. Al ritorno dalla funzione, la funzione `main` scrive 0 in SPR, in modo da silenziare l'altoparlante, e quindi termina.

4 Hard disk

L'hard disk è un esempio di dispositivo *a blocchi*. Con questo termine ci si riferisce a dispositivi di memorizzazione in cui l'informazione è organizzata in unità relativamente grandi, dette appunto blocchi, che sono anche l'unità di trasferimento. Per esempio, negli hard disk comuni i blocchi sono tradizionalmente grandi 512 byte, e non è possibile leggere o scrivere meno di 512 byte per volta. Pur essendo dispositivi di memoria, dunque, non possono essere interfacciati direttamente con la CPU, che invece accede ad unità molto più piccole, e devono essere considerati dei dispositivi di ingresso/uscita: per poter accedere a informazioni contenute in un hard disk, il software deve tipicamente prima copiare i relativi blocchi in memoria centrale, ordinando il trasferimento tramite una interfaccia di I/O.

Internamente, gli hard disk si possono scomporre in un componente detto *drive*, che comprende i dischi, le testine di lettura/scrittura e i loro servomeccanismi, e un secondo componente detto *controllore*, che contiene la logica in grado di pilotare il drive in base agli ordini impartiti dal software. Controllore e drive sono tipicamente venduti insieme, in un unico dispositivo, che poi deve essere collegato al resto del computer con un cavo che, nei PC moderni, è di tipo seriale.

L'informazione è memorizzata magneticamente sulle facce dei dischi, ciascuna delle quali è suddivisa in tracce concentriche. Ciascuna traccia è poi suddivisa in settori. Il numero di settori per traccia aumenta man mano che ci si sposta dal centro verso la periferia del disco, in modo che la dimensione fisica dei settori resti all'incirca costante. I dischi contenuti del drive (tipicamente 2 o poco più) sono impennati ad uno stesso asse centrale e sono tenuti costantemente in rotazione (salvo essere messi a riposo in caso di inattività prolungata, per risparmiare energia), con velocità angolari che possono andare da qualche migliaio a qualche decina di migliaia di rpm (rotazioni per minuto), a seconda

del modello. Le testine di lettura/scrittura, una per ciascuna faccia di ogni disco, sono anch'esse solidali ad uno stesso asse di rotazione, in modo che in ogni istante si trovino tutte in corrispondenza della stessa traccia su ogni faccia. Le tracce che si trovano alla stessa distanza dal centro su ogni faccia, e che dunque possono essere lette/scritte senza spostare le testine, formano un cosiddetto *cilindro*. Ogni settore è dunque identificato geometricamente da tre coordinate: il numero della faccia (o, equivalentemente, della testina), il numero della traccia sulla faccia (o, equivalentemente, il numero del cilindro), il numero del settore all'interno della traccia. Per poter riferire un settore, il drive ha bisogno di conoscere queste tre coordinate. A quel punto sposterà (se necessario) le testine per portarle sul cilindro richiesto (spendendo un tempo detto di *seek*), attiverà la testina corrispondente alla faccia richiesta e aspetterà che il settore richiesto, per effetto della rotazione costante dei dischi, vi passi di sotto (spendendo un ulteriore tempo detto *latency*). Solo a questo punto il settore può essere letto o scritto. In ogni caso il trasferimento dei dati avverrà da/verso un buffer interno, da cui poi dovrà essere trasferito in memoria centrale in qualche altro modo (come vedremo negli esempi più avanti).

Si noti che i tempi di seek e latency, essendo legati a fattori meccanici, sono migliorati nel tempo, ma molto più lentamente rispetto alla velocità delle componenti elettroniche, soprattutto del processore. In particolare, seek e latency sono entrambi dell'ordine di qualche millisecondo, contro le frazioni di nanosecondo del ciclo di un tipico processore. Questo comporta che, in un sistema moderno, è bene cercare di leggere/scrivere da settori contigui, in modo da minimizzare lo spostamento delle testine e ridurre i tempi di latenza. In pratica, oggi, conviene usarli come dispositivi sequenziali, anche se, quando furono introdotti dall'IBM negli anni '50, avevano proprio lo scopo di permettere l'accesso casuale e superare i limiti delle unità a nastro magnetico. Oggi, soprattutto nell'elettronica di consumo, sono stati praticamente sostituiti dagli hard disk a stato solido (SSD), che non hanno parti in movimento e hanno tempi di accesso nell'ordine dei microsecondi. Sono ancora comunque molto usati nei *data centre*, perché il loro prezzo per bit è ancora ordini di grandezza inferiore a quello degli SSD, e probabilmente sarà ancora così per molti anni.

4.1 Interfaccia per il programmatore

Il software interagisce con l'hard disk solo attraverso una interfaccia che, a sua volta, interagisce con il controllore a bordo dell'hard disk. Tramite i registri dell'interfaccia il software impedisce i comandi di lettura o scrittura, specificando quale settore deve essere coinvolto, e accede al buffer interno, sia per estrarne il contenuto dopo la fine di una operazione di lettura, sia per riempirlo con i dati da memorizzare prima dell'inizio di una operazione di scrittura.

Facciamo qui riferimento all'interfaccia ATA (AT Attachment), che standardizza l'interfaccia che si trovava nel PC AT. Questa interfaccia prevede diversi registri a 8 bit e uno a 16 bit, tutti mappati nello spazio di I/O. Ci interessano per il momento soltanto i seguenti:

- SNR (Sector Number), CNL (Cylinder Nymber Low), CNH (Cylinder Number High), HND (Head aNd Drive): tutti da 8 bit, permettono di specificare il (primo) settore coinvolto nell'operazione;
- SCR (Sector Counter, 8 bit): permette di specificare quanti settori (in sequenza a partire dal primo) sono coinvolti nell'operazione;
- CMD (CoMmanD, 8 bit): permette di specificare il tipo di operazione (per es., lettura o scrittura);
- BR (Buffer Register, 16 bit): permette di accedere al buffer interno, due byte alla volta;
- STS (STatus, 8 bit): contiene due flag che permettono di sapere se la precedente operazione si è conclusa (e dunque è possibile accedere al registro BR);

Anche se l'interfaccia prevede che il programmatore identifichi un settore dando le coordinate geometriche (testina, cilindro e settore), questo ha ormai soltanto un interesse storico, in quanto l'organizzazione interna dei dischi è oggi molto più complessa di un tempo. Quello che faremo è di usare il modo di indirizzamento detto LBA (Logical Block Address), in cui ogni settore è identificato da un numero sequenziale, a partire da 0 fino al numero di settori contenuti nell'hard disk. Questo numero va scomposto in quattro parti e scritto nei registri SNR, CNL, CNH e nei 4 bit meno significativi di HND, perché i 4 bit più significativi di questo registro hanno altre funzioni (tra cui, abilitare LBA). Si noti che in questo modo si dispone di 28 bit per identificare un settore; con settori di 512 byte questo limita la dimensione massima di un hard disk a $2^{28} \times 2^9 = 2^{37}$ byte, cioè 128 GiB, che è poco per un hard disk moderno. Per gli hard disk più grandi, lo standard prevede che il numero di settore sia spezzato in due parti di 28 e 20 bit (modalità LBA48) e scritto in due passaggi negli stessi registri, con un meccanismo simile a quello visto per il timer. L'interfaccia permette di ordinare il trasferimento di più settori consecutivi, specificandone il numero nel registro SCR. In quel caso il controllore provverà ad incrementare automaticamente l'LBA dopo ogni trasferimento.

La macchina QEMU che usiamo per gli esempi fornisce un hard disk ATA, emulato dal file CE/share/hd.img nella nostra home directory. La cartella `esempio` contiene due programmi che mostrano come si può ordinare una scrittura (`hard-disk-1`) o una lettura (`hard-disk-2`) da questo hard disk. Entrambi i programmi sono strutturati in modo simile. La funzione `hd_set_lba()` accetta un LBA come parametro, lo spezza in quattro parti e lo scrive nei registri SNR, CNL, CNH e HND, abilitando anche il modo LBA. Il tipo `natl` è definito in `libce` come un **typedef** per `unsigned int` (32 bit). Per semplicità la funzione non abilita la doppia scrittura, quindi `lba` deve essere minore di 2^{28} . La funzione `hd_start_cmd(lba, quanti, cmd)` avvia un trasferimento di quanti settori a partire da quello di indirizzo `lba`. Il parametro `cmd` deve essere uno di quelli compresi dall'interfaccia. La libreria definisce le costanti `WRITE_SEC` e `READ_SECT`, che possono essere passate

alla funzione per ordinare operazioni rispettivamente di scrittura e lettura. La funzione si limita a trasferire i parametri nei corrispondenti registri (usando `hd_set_lba()` per impostare l'LBA). Una volta avviata l'operazione dobbiamo aspettare che il controllore ci dica che è possibile accedere al buffer interno (funzione `hd_wait_for_br()`) e quindi eseguire 256 scritture (o letture) in BR. Per fare quest'ultima operazione possiamo comodamente usare le istruzioni **outs** e **ins** con prefisso **rep**, che trasferiscono da (verso) la memoria all'indirizzo contenuto in **rsi** il numero di byte/word/long (a seconda del suffisso) specificato in **rcx**. La libreria `libce` contiene le funzioni `outputb*` e `inputb*` (dove l'asterisco sta per b, w o l) che utilizzano queste istruzioni. Si noti che l'operazione di controllo di STS e successiva scrittura/lettura in BR va ripetuta per ogni settore coinvolto nell'operazione.

La funzione `main` usa queste funzioni per scrivere o leggere un paio di settori a partire da un certo LBA, trasferendoli da/verso l'array `buff`. Il programma `hard-disk-2` mostra poi sul video il contenuto dell'array `buff` a lettura ultimata. Il programma `hard-disk-1` mostra solo il messaggio `OK` per dire che l'operazione si è conclusa. Possiamo verificare che la scrittura è stata effettivamente eseguita correttamente esaminando il contenuto del file che emula l'hard disk “fuori” dalla macchina virtuale, per esempio con il comando

```
hexdump -C ~/CE/share/hd.img
```

che dovrebbe mostrare una serie di caratteri ‘f’ a partire dall'offset 512 (200 in esadecimale), corrispondente all'inizio del settore con LBA pari a 1.

Interruzioni

G. Lettieri

2 Aprile 2022

Il processore che abbiamo visto fino ad ora esegue un flusso di istruzioni dettato da un unico programma: ogni istruzione del programma ordina al processore sia quali operazioni deve compiere, sia qual è l'istruzione successiva. In particolare, ogni routine del programma va in esecuzione solo perché è stata esplicitamente invocata da una istruzione precedente nel flusso sequenziale, che le ha trasferito “volontariamente” il controllo. Vogliamo ora aggiungere un meccanismo nuovo: il programmatore può associare l'esecuzione di una routine al verificarsi di un *evento*. Più in generale, il sistema definisce un insieme (finito) di eventi e_1, e_2, \dots, e_n e permette al programmatore di associarvi delle routine, chiamiamole r_1, r_2, \dots, r_n , con r_i associata all'evento e_i , per $1 \leq i \leq n$. Per programmare il sistema il programmatore deve ora scrivere un programma principale, chiamiamolo p , ma può anche scrivere le routine r_1, \dots, r_n e associarle agli eventi e_1, \dots, e_n (o un sottoinsieme di essi, in base alle proprie necessità). Il processore partirà eseguendo p e obbedendo alle istruzioni di p , in sequenza, esattamente come prima. Mentre esegue p , però, il processore controllerà se si è verificato uno degli eventi previsti e, in tal caso, salterà automaticamente alla routine associata, *interrompendo* il programma p . L'idea è che l'interruzione sia comunque momentanea e che, al termine della routine, l'esecuzione ritorni dal programma p , dal punto in cui era stato interrotto.

Per capire perché potrebbe servirci un meccanismo del genere, e a che tipo di eventi potremmo essere interessati, ispiriamoci ad un esempio che fece uno dei primi ricercatori che si occupò di studiare le interruzioni—Edsger Dijkstra¹. Immaginiamo di dover scrivere un programma che deve calcolare i valori di una funzione $f(x)$ per vari valori di x (il tipico problema per cui i computer sono stati inventati), in modo da stampare una tabella dei valori della funzione:

x	$\sin(x)$
0.1000	0.0998
0.1001	0.0999
0.1002	0.1000
...	

Per guadagnare tempo, ci piacerebbe che, mentre è in corso la stampa dell'ultimo valore calcolato, per esempio per $x = 0.1001$, il nostro programma

¹L'esempio originale si trova qui: <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD13xx/EWD1303.html>.

potesse andare avanti a calcolare il successivo valore, per $x = 0.1002$. In questo modo stampa e calcoli procederebbero in parallelo. Come possiamo fare? Supponiamo di avere un dispositivo di uscita (una stampante) con i due classici registri TBR (Transmit Buffer Register) e un registro di stato STS. Per stampare un carattere ne dobbiamo scrivere il codice ASCII nel TBR. La stampante impiega un certo tempo per stampare il carattere, e per tutto quel tempo non può accettare un nuovo carattere in TRB. Un bit READY del registro di stato ci dice quando la stampante è pronta a ricevere un nuovo carattere. Il nostro programma dovrebbe essere organizzato così: quando ha finito di calcolare un valore di $f(x)$, prepara la riga da stampare in un buffer in memoria, aspetta che la stampante sia pronta (legge ripetutamente STS) e poi scrive il primo carattere. A questo punto, invece di aspettare che la stampante sia pronta a ricevere il prossimo carattere, prosegue con il calcolo del secondo valore. Qui viene la difficoltà: la routine che esegue i calcoli, ogni tanto, deve leggere STS e, se trova la stampante pronta, deve scrivere in TBR il prossimo carattere preso dal buffer, poi proseguire con i calcoli. Ogni quanto bisogna inserire le istruzioni che controllano STS? Se lo si fa troppo spesso si rallenta il calcolo del prossimo valore, se lo si fa troppo raramente si rallenta la stampa. Anche se troviamo una frequenza ideale, potrebbe non esserci una soluzione ottima: che succede se nella routine che fa i calcoli c'è un lungo ciclo con un corpo di poche istruzioni?

```
// controllo STS?
for (int i = 0; i < 1000000; i++) {
    // controllo STS?
    v[i]++;
}
// controllo STS?
```

Inserendo il controllo di STS solo fuori dal ciclo si rischia di far passare troppo tempo tra un controllo e il successivo, ma se lo si inserisce dentro il ciclo si rischia di farlo inutilmente troppo spesso. Che succede poi se il programma che fa i calcoli usa una funzione di libreria già scritta, magari da qualcun altro? Questa sicuramente non conterrà già i controlli di STS, e dovremmo crearne una versione modificata. A questi problemi aggiungiamo anche il fatto che il programma e le librerie diventano presto illegibili se dobbiamo inserire i controlli di STS in mezzo a funzioni che fanno tutt'altro.

1 Interruzioni (singola sorgente)

Per risolvere questi problemi modifichiamo leggermente l'hardware. L'idea di base (da perfezionare) è di portare il bit READY di STS direttamente in ingresso alla CPU (Figura 1), quindi di modificare il microprogramma della CPU in modo che, dopo ogni fase di esecuzione, controlli lo stato del nuovo ingresso e, se lo trova attivo, carichi un nuovo valore in **rip**. L'effetto è di avere inserito automaticamente un salto nel programma nel momento in cui la stampante diventa pronta. Visto che il controllo di READY viene eseguito dopo ogni

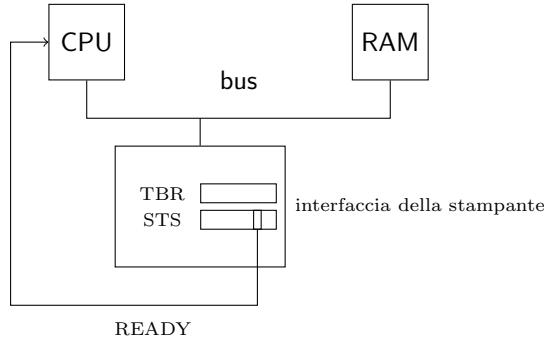


Figura 1: Idea di base del meccanismo delle interruzioni.

istruzione, non rischiamo di farlo troppo raramente (almeno rispetto ai tempi di una stampante); visto che abbiamo realizzato il controllo con un test in hardware di un piedino di ingresso, non paghiamo il costo di dover eseguire le istruzioni che leggono STS e controllano READY; visto che il controllo è fatto automaticamente dalla CPU, il programma principale non deve più contenere queste istruzioni “estranee” che ne compromettono la leggibilità. Il sistema è ora in grado di riconoscere autonomamente il verificarsi dell’evento *e* che corrisponde a “la stampante è pronta a ricevere il prossimo carattere”. Il programmatore deve solo scrivere una routine *r* che invia il prossimo carattere da stampare e associarla ad *e*. Il modo in cui avviene questa associazione, nel caso in cui ci sia un unico evento possibile, può essere molto semplice: il processore può caricare un indirizzo costante in **rip** quando riconosce l’evento, e il programmatore può semplicemente caricare la sua routine *r* a quell’indirizzo. Il programmatore poi scrive il programma principale *p* che ciclicamente calcola un nuovo valore, prepara la riga da stampare nel buffer e passa a calcolare il valore successivo. Periodicamente, il programma *p* verrà *interrotto* per eseguire la routine *r* che stampa il prossimo carattere preso dal buffer.

Un modo per concettualizzare ciò che avviene, dal punto di vista della stampante, è che questa ha bisogno di “attenzione” da parte del software quando ha finito di stampare un carattere, perché il software le deve dire cosa stampare dopo. Inoltre dunque una richiesta di attenzione, o di *servizio*, alla CPU, settando il bit READY. La CPU reagisce a questa richiesta interrompendo il programma principale (per questo la richiesta della stampante viene anche detta *richiesta di interruzione*) e cedendo forzatamente il controllo ad una *routine di servizio* per la stampante. Quando questa routine scrive il prossimo carattere in TBR, la stampante (la sua interfaccia) pone READY a zero, e questo “rimuove” la richiesta di servizio dalla CPU, a significare che la stampante ha ricevuto il servizio che aveva richiesto. La scrittura in TBR, dunque, funge da *risposta* alla richiesta di interruzione.

Questa è l’idea di base, che però richiede di essere perfezionata perché possa

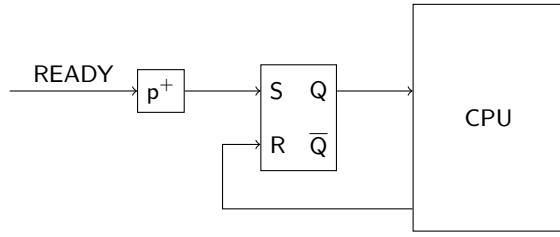


Figura 2: Rete per il riconoscimento di una nuova richiesta di interruzione.

funzionare effettivamente.

- A livello hardware, dobbiamo intanto assicurarci che la CPU non veda due volte la stessa richiesta. Per come abbiamo descritto il meccanismo, infatti, quello che accade è che, non appena la CPU vede READY 1, salta alla prima istruzione della routine di servizio. Se questa istruzione non è esattamente quella che scrive in TBR, alla fine della sua esecuzione READY sarà ancora a 1 e, dunque, la CPU salterà nuovamente alla prima istruzione della routine di servizio, ciò quella appena eseguita, e così all’infinito. Questo problema può essere risolto in hardware con una rete come in Figura 2, in cui la richiesta viene trasformata in un impulso che setta un latch S/R. Il microprogramma della CPU provvederà a resettare il latch dopo aver visto la richiesta, in modo che il latch sia di nuovo attivo solo se arriva una *nuova* richiesta di interruzione.
- Se vogliamo che la routine di servizio possa tornare al programma principale nel punto in cui era stato interrotto, la CPU non può limitarsi a sovrascrivere **rip** quando accetta la richiesta, ma deve prima scrivere il precedente valore da qualche parte. Una soluzione è di scriverlo sullo stack, in modo che la routine di servizio possa terminare con una **ret** (o, come vedremo, con una istruzione simile); questa è la soluzione adottata nei processori Intel.
- Anche a livello software vanno risolti molti problemi, per poter scrivere correttamente il programma principale e la routine di servizio. In realtà, i problemi sono così tanti che Dijkstra, non a torto, ha scritto che questo semplice meccanismo aprì un “vaso di Pandora”. Per il momento vi accenniamo soltanto—avremo modo di ritornarci.

Risolti questi problemi, possiamo pensare di raffinare ed estendere il meccanismo in vari modi. Il problema principale, dal punto di vista del software, è l’imprevedibilità del momento in cui può essere accettata una richiesta, con conseguente salto alla routine di servizio. Dal punto di vista del programmatore è utile poter temporaneamente “disabilitare” le richieste di interruzione, fare cioè in modo che la CPU le ignori durante l’esecuzione di certe porzioni di codice. La soluzione tipicamente adottata, anche nei processori Intel, è che la CPU abbia

un flag di *interrupt enabled* che possa essere settato e resettato via software, e che la CPU ascolti le richieste di interruzione solo se questo flag è settato. Nel processore intel questo è il flag IF (Interrupt Flag, bit 9) del registro dei flag e le istruzioni `cli` e `sti` possono essere usate per resettarlo e settarlo, rispettivamente. Le richieste che dovessero arrivare mentre IF è zero verrano servite non appena IF tornerà a 1.

Una porzione di codice che non vogliamo venga interrotta è, tipicamente, la routine di servizio stessa, cosa che invece può accadere non appena la routine esegue l'istruzione di risposta alla richiesta di servizio (la scrittura in TBR, nel nostro esempio). La situazione è talmente tipica che il processore Intel può (su richiesta del programmatore) resettare automaticamente il flag IF ogni volta che accetta una richiesta. Prima di farlo, il processore salva in pila il vecchio valore del registro dei flag, oltre al vecchio valore di `rip`. Il processore fornisce poi l'istruzione `iretq` che preleva dalla pila sia `rip` che i flag. Le routine di servizio devono terminare con `iretq`, invece che con `ret`, in modo da estrarre dalla pila entrambi i valori (e altri che vedremo) e riportare IF a 1.

2 Interruzioni (più sorgenti)

Un altro modo di estendere il meccanismo è quello di prevedere più sorgenti di interruzione, in modo che il sistema possa riconoscere un certo numero di eventi distinti e il programmatore possa associare una routine di servizio diversa a ciascuno di questi eventi, come avevamo detto all'inizio. Delle periferiche che abbiamo studiato, tre sono in grado di generare richieste di interruzione:

- la tastiera, quando RBR è pieno o TBR si svuota dopo una scrittura;
- il timer, quando il contatore 0, opportunamente programmato, termina il conteggio;
- l'hard disk, quando il buffer interno è pieno dopo una operazione di lettura o vuoto dopo una operazione di scrittura.

La tastiera e l'hard disk generano richieste di interruzione solo se queste sono state abilitate *nell'interfaccia*. Questa è un'altra soluzione molto tipica, che permette al programmatore di non dover gestire richieste di interruzione da parte di interfacce che non sta usando. Le interfacce prevedono dunque uno o più registri di *controllo*, con varie funzioni tra cui abilitare o disabilitare l'interfaccia a generare richieste di interruzione (consultare gli esempi di I/O per i dettagli).

Per supportare più sorgenti di richieste di interruzione dobbiamo rispondere a tre domande:

- come inoltrare le richieste alla CPU;
- se e come associare una routine di servizio diversa ad ogni sorgente;
- cosa fare in caso di richieste che arrivano mentre ancora è in corso il servizio di una richiesta precedente da parte di un'altra sorgente.

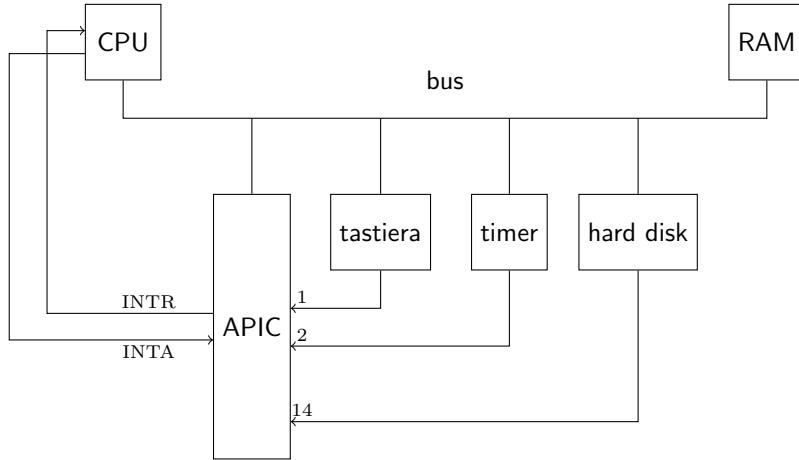


Figura 3: Controllore delle interruzioni.

2.1 Controllore delle interruzioni

Per rispondere alla prima domanda, teniamo presente che il meccanismo che abbiamo introdotto non ha cambiato la natura fondamentalmente sequenziale della CPU: la CPU, dal suo punto di vista, continua ad avere un unico flusso di controllo e ad ogni istante esegue un unico programma: il programma principale, oppure una routine di servizio². Anche se abbiamo più sorgenti di richieste di interruzione e più routine di servizio, la nostra CPU potrà comunque eseguirne solo una alla volta. Ha allora senso che la CPU abbia un unico piedino su cui riceve richieste di interruzione, e un qualche componente esterno si preoccupi di raccogliere tutte le richieste provenienti dalle varie sorgenti, ordinarle in qualche modo e inoltrarle una alla volta alla CPU³. Questo componente è detto *controllore delle interruzioni*. Lo schema del sistema è ora quello di Figura 3. In particolare, facciamo riferimento al controllore APIC (Advanced Programmable Interrupt Controller) che si trova nei sistemi basati su chipset Intel e anche, emulato, nella nostra macchina QEMU. L'APIC possiede 24 piedini di ingresso, numerati da 0 a 23, a cui possono essere collegate le linee di richiesta di interruzione di altrettante interfacce. In figura vediamo come sono collegate le periferiche della macchina QEMU. Il controllore permette la gestione *vettorizzata* delle richieste di interruzione. Con questo termine ci si riferisce al fatto che

²La CPU, in realtà, non è neanche consapevole della distinzione: per lei, la gestione della richiesta di interruzione termina subito dopo aver effettuato il salto, dopo di che riprende il normale ciclo di prelievo, decodifica, esecuzione, a partire dal nuovo `rip`. In particolare, la CPU non è “in attesa” che venga eseguita una `iretq` dopo l'accettazione di una richiesta di interruzione, come non è in attesa che venga eseguita una `ret` dopo una `call`.

³Anche se la CPU ha più di un piedino di ingresso per le richieste di interruzione, serve comunque una circuiteria, interna alla CPU stessa ma concettualmente distinta, che le raccolga e ne lasci passare una sola per volta. Vedremo che, in effetti, la CPU Intel, come molte altre, ha anche un altro piedino da cui riceve richieste di interruzione “non mascherabili”: queste hanno precedenza su tutte le altre.

ad ogni sorgente di interruzione è associato un *tipo* numerico, che viene passato alla CPU insieme alla richiesta. La CPU utilizza questo tipo per consultare una tabella che associa i tipi agli indirizzi delle routine di servizio. Nei processori Intel la tabella si chiama Interrupt Descriptor Table (IDT) e il programmatore la può allocare ovunque in memoria, quindi caricarne l'indirizzo nel registro IDTR del processore. In questo modo è possibile associare una routine di servizio diversa ad ogni sorgente: basta scrivere l'indirizzo della routine nell'entrata della IDT corrispondente al tipo da associare. In assenza della vettorizzazione, ad ogni richiesta andrebbe in esecuzione sempre la stessa routine, che a quel punto dovrebbe capire via software quale interfaccia ha richiesto il servizio (per esempio, consultando i registri di stato di ogni interfaccia).

Nell'APIC l'associazione tra sorgente e tipo è configurabile dal programmatore. Il controllore possiede infatti un diverso registro interno, accessibile via software, per ognuno dei piedini di ingresso. Per ogni piedino è possibile specificare il tipo, su 16 bit, e altre informazioni legate a come l'APIC dovrà riconoscere le richieste in ingresso. In particolare, per ogni piedino è possibile decidere:

- se il riconoscimento di una nuova richiesta deve avvenire sul fronte (segnale in ingresso che cambia livello) o sul livello (vedere più avanti);
- se il segnale di ingresso deve essere considerato attivo quando è alto oppure quando è basso;
- se le richieste devono essere ignorate o meno (mascherate).

Per il dialogo con il processore supponiamo che ci siano due collegamenti: uno, INTR (INTerrupt Request), dall'APIC alla CPU, tramite il quale l'APIC inoltra una richiesta di interruzione; un altro, INTA (INTerrupt Acknowledge), dalla CPU all'APIC, usato per un protocollo di handshake:

1. quando l'APIC vuole inviare una richiesta di interruzione, attiva INTR e aspetta che sia attivo INTA;
2. la CPU controlla lo stato di INTR dopo l'esecuzione di ogni istruzione, se IF è 1; quando lo trova attivo, attiva a sua volta INTA;
3. quando l'APIC vede INTA attivo, passa il tipo dell'interruzione al processore sul bus e disattiva INTR⁴
4. quando il processore vede INTR disattivo, legge il tipo e disattiva INTA, chiudendo l'handshake.

⁴Restiamo sul vago su come il tipo viene passato, in quanto il vero APIC è in realtà scomposto in due parti: un I/O APIC sulla scheda madre e un Local APIC all'interno della CPU. Il dialogo, compreso lo scambio del tipo, avviene tra i due APIC su un bus dedicato. Questa organizzazione è pensata per i sistemi con più CPU, in modo da avere, per esempio, un unico I/O APIC che dialoga con tanti Local APIC, uno per CPU. Nel nostro caso possiamo semplificare la discussione, perché ci limitiamo al caso con una sola CPU.

L'APIC possiede altri tre registri interni che ci servono per capire in dettaglio il suo funzionamento: IRR (Interrupt Request Register), ISR (In Service Register) e EOI (End Of Interrupt). I registri IRR e ISR sono entrambi da 256 bit, un bit per ogni possibile tipo. Supponiamo per il momento che una sola interfaccia sia abilitata a inviare richieste di interruzioni, e chiamiamo i il piedino di ingresso dell'APIC a cui tale interfaccia è collegata. Supponiamo anche che il piedino i sia impostato per il riconoscimento sul fronte, e il tipo associato sia t . L'APIC svolge continuamente le seguenti azioni:

1. ad ogni fronte su i , se il bit t di IRR non è già attivo, lo attiva
2. se il bit t di IRR è attivo e il bit t di ISR non lo è, l'APIC inizia l'handshake con la CPU;
3. quando l'handshake arriva al punto 3 (la CPU ha accettato la richiesta), invia il tipo t e “sposta” il bit attivo da IRR in ISR (disattiva t in IRR e lo attiva in ISR);

Lo scopo di ISR è di tenere traccia di quali richieste sono attualmente in servizio sulla CPU. L'APIC assume che la CPU, nel momento in cui accetta la sua richiesta, passi ad eseguire la routine di servizio corrispondente, e per questo setta il bit t di ISR a 1. L'APIC non inoltrerà altre richieste dello stesso tipo fino a quando si troverà in questo stato (si veda il punto 2 qui sopra: l'handshake parte solo se il bit t di ISR è a zero). Un eventuale altro fronte sul piedino i verrà comunque registrato in IRR (punto 1). In pratica è come se i due bit t in IRR e ISR realizzassero una coda di due richieste: una in servizio e l'altra in attesa. Una terza richiesta che dovesse arrivare in questo stato, però, verrebbe persa. È compito del software far sapere all'APIC che la routine di servizio è terminata, scrivendo un valore qualsiasi nel registro EOI. In risposta a questa scrittura l'APIC azzera il bit t in ISR e, se il bit t di IRR è 1 (c'è un'altra richiesta pendente arrivata nel frattempo) si riporta al punto 2, altrimenti al punto 1.

Nel caso di riconoscimento sul livello l'APIC si comporta diversamente solo per quanto riguarda il riconoscimento di una nuova richiesta sul piedino: se il bit t di IRR è a zero e c'è un fronte su i , l'APIC regista una nuova richiesta, come nel punto 1 qui sopra; da questo punto in poi, però, l'APIC smette di osservare il piedino i ; lo osserverà di nuovo solo all'arrivo del prossimo EOI: se in quel momento lo ritrova ancora (o di nuovo) attivo, regista una nuova richiesta. L'idea è che normalmente la periferica dovrebbe aver disattivato il piedino i , una volta che la routine di servizio ha effettuato la risposta alla richiesta (si pensi all'esempio della stampante: la richiesta resta attiva fino a quando la routine non scrive in TBR) e il software dovrebbe inviare l'EOI *dopo* che la routine ha effettuato questa azione. Dunque, se la richiesta è ancora attiva, deve trattarsi di una nuova richiesta.

Negli esempi vedremo quando conviene scegliere un riconoscimento sul fronte o uno sul livello.

2.2 Gestione annidata delle richieste di interruzione

Passiamo ora a considerare il caso generale, in cui le richieste possono arrivare da qualsiasi piedino, anche contemporaneamente. In generale si preferisce realizzare la *gestione annidata* delle richieste di interruzione, in cui una routine di servizio può essere a sua volta interrotta da un'altra routine di servizio. La cosa ha senso se la seconda routine ha una *priorità* maggiore rispetto alla prima (per esempio, le richieste che arrivano dal timer hanno in genere massima priorità, in quanto non è possibile ritardarle). Ovviamente l'annidamento può avvenire a qualsiasi livello, con la seconda routine che può essere interrotta da una terza, a maggiore priorità, e così via. Dal momento che tutti gli indirizzi a cui ritornare sono salvati sulla stessa pila, le routine di servizio devono poi terminare in ordine LIFO: l'ultima ritorna alla penultima e così via.

Affinchè l'annidamento sia possibile, la CPU non deve disabilitare le interruzioni mentre è in esecuzione una routine di servizio. Nel processore Intel questo può essere ottenuto facilmente, in quanto la disabilitazione delle interruzioni è facoltativa (basta settare un flag nelle entrate della IDT corrispondenti alle routine che si vuole eseguire in questo modo).

L'APIC può essere usato per aiutare nella gestione annidata delle interruzioni con priorità. L'APIC, infatti, interpreta il tipo associato ad un piedino come la priorità delle richieste provenienti da quel piedino. Più precisamente, i 4 bit più significativi del tipo rappresentano, per l'APIC, la sua *classe di priorità*, in ordine numericamente crescente (15 è dunque la classe di priorità massima). Quando una nuova richiesta viene registrata in IRR, l'APIC confronta la classe di priorità della richiesta più a sinistra in IRR (la richiesta pendente a priorità maggiore), sia p_r , con la classe di priorità della richiesta più a sinistra in ISR (la richiesta in servizio a priorità maggiore), sia p_s , e inizia l'handshake con la CPU se, e solo se, $p_r > p_s$. Quando la CPU accetta questa richiesta l'APIC si comporta come al solito, spostando il bit da IRR in ISR. Si noti che ora più bit di ISR possono essere a 1, riflettendo il fatto che più routine di servizio hanno iniziato la propria esecuzione e non sono ancora concluse. Si noti, inoltre, che il fatto che l'APIC non inoltra le richieste con $p_r \leq p_s$ comprende il caso particolare, che abbiamo discusso in precedenza, in cui arrivi una nuova richiesta dallo stesso piedino di una richiesta già accettata, perché in quel caso le classi di priorità sono uguali.

Quando il software scrive in EOI, l'APIC resetta il bit più a sinistra in ISR. In pratica, l'APIC assume che il software stia rispettando l'ordine LIFO di esecuzione delle routine di servizio, e dunque che la scrittura sia stata effettuata perché è terminata la routine a più alta priorità tra quelle attualmente attive. Dopo aver resettato il bit, l'APIC ricontrolla le classi di priorità in IRR e in ISR per vedere se è necessario inoltrare una nuova richiesta.

Si noti che l'APIC, per decidere se inoltrare o no una richiesta, guarda esclusivamente le classi di priorità, ignorando i 4 bit meno significativi dei tipi. Al momento di inoltrare una nuova richiesta alla CPU, però, usa l'intero tipo per scegliere quale richiesta inviare tra quelle registrate in IRR.

Eccezioni

G. Lettieri

20 Aprile 2018

Le prime 32 entrate della Interrupt Descriptor Table sono riservate per le *eccezioni*. Con questo termine ci si riferisce a condizioni di errore o speciali che il processore rileva mentre sta eseguendo le normali istruzioni. Per esempio, una operazione di divisione (div o idiv) in cui il divisore è zero causa una eccezione di “divisione per zero”. Il processore tratta le eccezioni in modo molto simile alle interruzioni esterne: vi associa un tipo numerico (compreso tra 0 e 31) e lo usa per accedere ad un gate della IDT, dove trova l’indirizzo di una routine a cui saltare. I tipi delle eccezioni sono fissi e consultabili sul manuale del processore. Per esempio, l’eccezione di divisione per zero ha tipo 0. Il meccanismo di salto è del tutto simile a quello delle interruzioni esterne, con il salvataggio in pila di informazioni analoghe. In particolare, anche le routine di gestione delle eccezioni, se vogliono tornare al programma principale, devono farlo con una istruzione **iretq** e non con una semplice **ret**.

Le eccezioni sono classificabili ulteriormente come segue:

trap vengono sollevate solo tra l’esecuzione di una istruzione e la successiva;

fault vengono sollevate *durante* l’esecuzione di una istruzione;

abort possono verificarsi in qualsiasi momento e indicano errori particolarmente gravi.

Per le eccezioni di tipo fault il processore salva in pila l’indirizzo dell’istruzione che stava eseguendo mentre ha rilevato il fault, e non l’indirizzo dell’istruzione successiva come per i trap e le interruzioni esterne. L’idea è che a volte la routine di gestione dell’eccezione può correggere la condizione che ha causato il fault e poi ritornare (con la **iretq** finale) a *rieseguire* l’istruzione, che questa volta non dovrebbe più generare il fault o al massimo dovrebbe generarne uno diverso, per qualche altra condizione di errore.

Si noti che, per rendere possibile la riesecuzione di una istruzione che era stata precedentemente eseguita fino ad un certo punto (e dunque poteva aver già modificato qualche registro), il processore deve essere in grado di ritornare allo stato precedente l’inizio della prima esecuzione. Per quanto riguarda lo stato dei registri, il processore può apportare tutte le modifiche in delle copie di lavoro, trasferendo il contenuto dalle copie ai veri registri solo alla fine dell’esecuzione di ogni istruzione. Se si verifica un fault durante l’esecuzione, è sufficiente ignorare

```

1 int main()
2 {
3     int x = 3, y = 0;
4     return x / y;
5 }
```

Figura 1: Un programma che causa una eccezione di “divisione per zero”.

le copie di lavoro per ritornare allo stato precedente. Il caso delle istruzioni che scrivono in memoria va invece esaminato istruzione per istruzione, ma in generale non ci sono problemi se si esegue una sola scrittura alla fine dell'esecuzione (che è il caso più comune).

La libreria `libcex`, che abbiamo usato fino ad ora per tutti gli esempi, inizializza la IDT in modo che ogni eccezione causi un salto ad una funzione (della libreria stessa) che stampa un messaggio di errore e blocca il processore eseguendo l'istruzione `hlt` (si veda la funzione `init_idt` nel file `64/init_idt.s`, seguendo la catena di chiamate fino a `gestore_eccezioni()`).

In Figura 1 troviamo un semplice programma che causa una eccezione. Si noti che il programma deve essere compilato senza ottimizzazioni, altrimenti il compilatore si accorge facilmente del tentativo di dividere per 0 e, dal momento che lo standard dichiara tale operazione come “indefinita”, può tradurre l'operazione in qualunque modo. In questo caso, molto probabilmente, si limiterebbe ad eliminare l'operazione di divisione. Scriviamo il codice di Figura 1 in un file di nome `prova.cpp` e compiliamolo con lo script `compile1`. Possiamo controllare che il compilatore abbia effettivamente generato l'istruzione di divisione osservando l'output del comando

```
objdump -d | grep idivl
```

Dovremmo ottenere qualcosa di simile:

```
20015d: f7 7d f8           idivl    -0x8 (%rbp)
```

Il primo numero è l'indirizzo a cui si troverà l'istruzione quando il programma verrà caricato.

Se lanciamo ora l'eseguibile con lo script `boot` dovremmo vedere che la macchina virtuale parte e si ferma, scrivendo sulla console un messaggio di errore. Il messaggio mostra il tipo dell'eccezione (0) e l'instruction pointer salvato in pila. In questo caso dovrebbe coincidere con `0x20015d`, perché l'eccezione 0 è di tipo fault.

Per vedere la differenza con le eccezioni di tipo trap proviamo a generare l'eccezione di *break-point*, che ha tipo 3. Questa eccezione è sollevata dall'istruzione `int3` alla fine della sua esecuzione. Essendo di tipo trap, il processore deve salvare in pila l'indirizzo dell'istruzione successiva. Dopo aver copiato il codice di

¹Come per tutti gli esempi che usano gli script `compile` e `boot`, dobbiamo trovarci in una directory che contiene inizialmente solo i file `*.cpp` e `*.s` dell'esempio.

```

1 int main()
2 {
3     int x = 0;
4     x++;
5     asm("int3");
6     x++;
7     return x;
8 }
```

Figura 2: Un programma che causa una eccezione di “break-point”.

Figura 2 in un file `prova.cpp` e averlo compilato con lo script `compile`, prendiamo nota dell’indirizzo dell’istruzione `int3` e delle istruzioni che le stanno attorno, con il comando:

```
objdump -d | grep -C 1 int3
```

Dovremmo ottenere un output del genere:

```

200152: 83 45 fc 01          addl    $0x1,-0x4(%rbp)
200156: cc                   int3
200157: 83 45 fc 01          addl    $0x1,-0x4(%rbp)
```

Se ora lanciamo il programma sulla macchina virtuale questa si ferma nuovamente con un messaggio di errore, questa volta per l’eccezione di tipo 3. L’instruction pointer salvato deve essere quello dell’istruzione *successiva* alla `int3` (0x200157 nel nostro caso).

Per eseguire una routine di nostra scelta ogni volta che si verifica una eccezione, operiamo in modo del tutto analogo a come abbiamo fatto per le interruzioni esterne: prepariamo un gate della IDT in modo che punti alla nostra funzione. Questa volta, però, non possiamo scegliere il gate che preferiamo, ma dobbiamo usare quello relativo all’eccezione che vogliamo intercettare. Per esempio, se vogliamo intercettare le eccezioni di divisione per zero dobbiamo usare il gate 0, e se vogliamo intercettare i break-point dobbiamo usare il gate 3.

Come esempio intercettiamo le eccezioni di tipo 1. Queste sono eccezioni di *debug* generate, tra le altre cose, dal meccanismo del *Single Step* (esecuzione passo passo). Tale meccanismo viene attivato settando il flag TF del registro dei flag. Quando tale flag è attivo il processore genera una eccezione di debug dopo aver eseguito *ogni istruzione*. Si tratta di una eccezione di trap, quindi l’instruction pointer salvato è quello dell’istruzione successiva (ancora da eseguire).

Il programma in Figura 3 associa la funzione `a_debug` al gate numero 1 (linea 14). La funzione `a_debug` è definita in Figura 4 e si limita a chiamare la funzione `c_debug` passandole il valore dell’instruction pointer salvato in pila dal processore (linea 17). Si noti che la funzione salva e ripristina tutti i registri, perché verrà chiamata dopo ogni istruzione del programma principale (se TF

```
1 #include <libce.h>
2
3 extern "C" void enable_single_step();
4 extern "C" void disable_single_step();
5 extern "C" void a_debug();
6 extern "C" void c_debug(void *rip)
7 {
8     flog(LOG_DEBUG, "rip=%p", rip);
9 }
10
11 int main()
12 {
13     int x = 0;
14     gate_init(1, a_debug);
15     enable_single_step();
16     x++;
17     x++;
18     x++;
19     x++;
20     disable_single_step();
21     pause();
22     return x;
23 }
```

Figura 3: Un programma che intercetta l'eccezione di debug, parte C++.

```

1 #include "libce.s"
2 .global enable_single_step
3 enable_single_step:
4     pushf
5         orw $0x0100, (%rsp)
6     popf
7     ret
8 .global disable_single_step
9 disable_single_step:
10    pushf
11        andw $0xFEFF, (%rsp)
12    popf
13    ret
14 .global a_debug
15 a_debug:
16     salva_registri
17     movq 120(%rsp), %rdi
18     call c_debug
19     carica_registri
20     iretq

```

Figura 4: Un programma che intercetta l'eccezione di debug, parte Assembler.

è settato). La macro `salva_registri` è definita nel file `libce.s` (nella libreria) e salva in pila tutti i registri generali tranne `rsp` (perché questo verrà ripristinato dal normale utilizzo della pila). Dunque, al momento di eseguire l'istruzione alla linea 17, l'instruction pointer si troverà 120 byte più in basso rispetto alla cima corrente della pila ($120 = 8 \times 15$).

La funzione `c_debug` (Figura 3) stampa il valore dell'instruction pointer utilizzando la funzione di libreria `flog()`, che invia il messaggio sulla porta seriale. Nel nostro caso abbiamo impostato la macchina virtuale affinché tutto ciò che viene inviato alla porta seriale venga mostrato sul terminale da cui abbiamo eseguito `boot`, quindi è lì che vedremo questi messaggi. La funzione `flog()` accetta un primo valore che indica il tipo di messaggio che si sta inviando (i possibili valori sono `LOG_DEBUG`, `LOG_INFO`, `LOG_WARN` e `LOG_ERR`). L'effetto è solo quello di cambiare le prime tre lettere della linea che viene inviata alla porta seriale, ma l'informazione potrebbe essere usata per filtrare i tipi di messaggi che si vuole o non si vuole vedere. Il secondo argomento è una stringa che verrà interpretata in modo simile a come opera la funzione `printf()` della libreria standard del C (la funzione si può usare anche in C++, includendo `<cstdio>`). La stringa rappresenta un modello per il messaggio che deve essere generato. Tutti i caratteri diversi da “%” verranno riprodotti così come sono, mentre i caratteri “%” rappresentano dei segnaposto per dei valori che devono essere inseriti in quel punto. Per ogni segnaposto deve essere passato

un ulteriore parametro (in ordine, dopo la stringa) e deve essere specificato, dopo il carattere “%”, in che modo il valore del parametro deve essere mostrato. Nel nostro caso usiamo il carattere “p” per specificare che vogliamo mostrare un puntatore (verrà stampato in esadecimale) e passiamo `rip` come parametro corrispondente. Altri caratteri possibili sono “d” per numeri da mostrare in base 10, “x” per numeri da mostrare in esadecimale, e “s” per stringhe di caratteri. I caratteri “d” e “x” richiedono parametri di tipo `int`, ma possono essere preceduti dal carattere “l” per usare parametri di tipo `long`.

La funzione `main()` abilita il Single Step (linea 15), esegue un po’ di istruzioni, e poi lo disabilita (linea 20). Le funzioni di abilitazione e disabilitazione sono scritte in Assembler (Figura 4). Si noti che non ci sono istruzioni apposite per modificare il flag TF, quindi utilizziamo le istruzioni `pushf` per salvare il contenuto del registro dei flag in pila, manipoliamo il valore in cima alla pila, e infine lo ricarichiamo nel registro dei flag con l’istruzione `popf`.

Compilando e avviando il programma dovremmo vedere un output del genere sulla console (non sul monitor della macchina emulata):

```
DBG 0 rip=00000000000200190
DBG 0 rip=00000000000200194
DBG 0 rip=00000000000200198
DBG 0 rip=0000000000020019c
DBG 0 rip=000000000002001a0
DBG 0 rip=000000000002001b8
DBG 0 rip=000000000002001b9
DBG 0 rip=000000000002001bf
DBG 0 rip=000000000002001c0
```

Se si va a guardare a cosa corrispondono i vari indirizzi (`objdump -dS`) si noteranno alcune cose.

- Il processore genera (o non genera) l’eccezione di debug alla fine di una istruzione, ma lo fa in base al valore che TF aveva subito prima di eseguirla. L’istruzione alla linea 6 di Figura 4 porta TF a 1, ma prima che iniziasse TF valeva 0. Quindi il processore non genera l’eccezione alla sua fine, ma solo alla fine dell’istruzione successiva, che parte quando TF è già 1. L’eccezione è di tipo trap, quindi l’instruction pointer salvato è quello dell’istruzione ancora successiva. Il primo `rip` stampato, dunque, dovrebbe essere quello della prima istruzione `x++` di `main()`. L’ultimo `rip` stampato dovrebbe essere invece quello dell’istruzione `ret` alla linea 13 di Figura 4. L’istruzione `popf` porta TF a 0, ma quando era partita TF era ancora 1, e dunque verrà generata ancora una eccezione di debug alla sua fine, salvando l’instruction pointer dell’istruzione successiva.
- Il processore non genera ulteriori eccezioni di Single Step mentre sono in esecuzione le funzioni `a_debug` e `c_debug`. Questo perché il flag TF viene automaticamente resettato ogni volta che si attraversa un gate della IDT. L’idea è che vogliamo eseguire passo passo un certo programma da

```

1 #include <libce.h>
2 void foo() {
3     printf("foo()\n");
4 }
5
6 int main()
7 {
8     foo();
9     pause();
10    return 0;
11 }

```

Figura 5: Esercizio 1

debuggare, ma non vogliamo eseguire passo passo anche il debugger stesso. Il vecchio valore di TF viene salvato in pila, insieme a tutti gli altri flag, e ripristinato dalla **iretq** che termina la routine di eccezione. Si noti come, in base alla regola precedente, non ci sarà una nuova eccezione subito dopo la **iretq**, ma solo dopo l'istruzione successiva, che è quello che vogliamo.

Esercizio 1

L'istruzione **int3** può essere usata da un debugger per inserire un break-point in un punto del programma da debuggare. La codifica dell'istruzione **int3** in linguaggio macchina è 0xCC, su un solo byte. Il debugger può dunque sostituire il primo byte dell'istruzione a cui ci si vuole fermare con 0xCC, e poi restituire il controllo al programma. Il programma ora esegue liberamente, ma il processore genererà una eccezione di break-point se l'esecuzione arriva alla **int3** così inserita. L'eccezione sarà intercetta dal debugger, che così riacquisirà il controllo del processore e potrà chiedere ulteriori istruzioni all'utente.

Vogliamo modificare il `main` di Figura 5 in modo da inserire un break-point all'inizio della funzione `foo()` che faccia saltare ad una funzione che invii il messaggio “breakpoint all'indirizzo *x*” (dove *x* è l'indirizzo della prima istruzione di `foo()`) su log. Dopo l'invio del messaggio l'esecuzione del programma deve riprendere normalmente.

Soluzione

La soluzione è in Figura 6. Alla riga 18 del file C++ associamo la funzione `a_debug` all'eccezione 3 (breakpoint), quindi sovrascriviamo il primo byte di `foo` con l'istruzione **int3** (riga 21). Per poter poi eseguire correttamente il programma, ci salviamo il byte che stiamo sostituendo (riga 20).

La chiamata a `foo()` (riga 23) causerà ora un salto alla `a_debug` e poi alla `c_debug()`. Questa invia il messaggio al log (riga 12) e poi ripristina il primo byte della `foo()` (riga 13). Si noti che **int3** è di tipo trap e dunque il

```

1 #include <libce.h>
2 void foo() {
3     printf("foo()\n");
4 }
5
6 natb old;
7 extern "C" void a_debug();
8 extern "C" void c_debug(void *rip)
9 {
10     natb *p = static_cast<natb*>(rip);
11     p--;
12     flog(LOG_DEBUG, "breakpoint_all'indirizzo_%p", p);
13     *p = old;
14 }
15
16 int main()
17 {
18     gate_init(3, a_debug);
19     natb *p = reinterpret_cast<natb*>(foo);
20     old = *p;
21     *p = 0xCC;
22
23     foo();
24     pause();
25     return 0;
26 }
```

```

1 #include "libce.s"
2 .global a_debug
3 a_debug:
4     salva_registri
5     movq 120(%rsp), %rdi
6     call c_debug
7     carica_registri
8     decq (%rsp)
9     iretq
```

Figura 6: Soluzione esercizio 1

processore salva in pila l'indirizzo dell'istruzione successiva. Per questo motivo dobbiamo sottrarre 1 a `rip` prima di usarlo (riga 11).

Sempre per questo motivo, la `a_debug` deve decrementare di 1 l'indirizzo in cima alla pila prima di eseguire la `iretq` che ritornerà al programma interrotto (riga 8 del file assembler). In questo modo possiamo eseguire l'istruzione che avevamo sostituito con `int3`.

Esercizio 2

Quando gdb raggiunge un breakpoint ridà il controllo all'utente. Se questo decide di continuare l'esecuzione (istruzione `c`), come fa gdb garantire che l'istruzione su cui era stato inserito il break-point venga ora eseguita, e allo stesso tempo che ci sia una nuova eccezione di break-point se il programma dovesse ripassare in seguito da quella stessa istruzione?

Protezione

G. Lettieri

26 Aprile 2021

Per capire perché abbiamo bisogno del meccanismo della protezione, partiamo da un esempio. Supponiamo di trovarci negli anni '50 o '60 del secolo scorso, quando un computer occupava l'area di una palestra ed una Università poteva averne uno o forse due. In questo periodo i computer venivano usati in modalità *batch*. Gli utenti—ricercatori o studenti—preparavano i programmi a casa, su fogli di carta, scrivendoli in linguaggio macchina o in FORTRAN. Portavano poi i loro fogli al centro di calcolo, dove alcuni impiegati potevano trascriverli su schede perforate. Ogni pacco di schede, contenente il programma di un utente, rappresentava un *job*. L'utente consegnava poi il suo job agli operatori del computer; in un secondo momento sarebbe dovuto ritornare a ritirare i risultati, tipicamente sotto forma di un tabulato stampato su carta.

Gli operatori, gli unici ad avere accesso alla sala del computer, aspettavano di avere un mazzo (*batch*) di job e poi lo caricavano sul lettore di schede del computer. Questo eseguiva i job uno alla volta, caricando automaticamente il prossimo job dopo aver terminato il precedente.

In questi sistemi si voleva massimizzare il numero di job completati ogni ora, sfruttando il costosissimo processore nel modo più efficiente possibile.

Supponiamo ora che il job dell'utente 1 debba caricare una lunga serie di dati da un nastro magnetico e decida di svolgere questa operazione a controllo di programma. Il costosissimo processore verrà così sprecato in un banale ciclo di istruzioni che legge ripetutamente i registri del controllore del nastro, per ricevere i dati e copiarli in memoria. La “vera” elaborazione del job 1, quella che ha davvero bisogno delle piene capacità della CPU, comincerà solo quando tutti i dati saranno stati trasferiti. Per gli operatori del computer sarebbe molto meglio se il controllore del nastro fosse programmato per una operazione in DMA. In questo modo, mentre i dati vengono trasferiti in memoria, si potrebbe utilizzare la CPU per cominciare ad eseguire il prossimo job del batch. Il controllore segnalerebbe poi il termine dell'operazione di trasferimento con una richiesta di interruzione. All'arrivo della richiesta si potrebbe ritornare al job 1.

Come realizziamo questo schema? Al di là di come si faccia a saltare da un job ad un altro, il problema è che non possiamo attenderci collaborazione da parte degli utenti stessi: l'utente del job 1 non ha interesse a cedere la CPU ad un altro job, e l'utente del job 2 non ha interesse a ridarla all'utente 1 quando arriva l'interruzione. Purtroppo, però, mentre è in esecuzione un certo programma, la CPU obbedisce a *quel* programma e dunque, nel nostro

caso, al volere degli utenti 1 e 2 e non al volere degli operatori. Gli operatori possono scrivere le necessarie routine: una per programmare il lettore del nastro in DMA e passare al prossimo job; un'altra per rispondere alla richiesta di interruzione ritornando al job precedente. Ma niente può costringere gli utenti ad usare queste routine. Il primo utente può non chiamare la prima routine degli operatori, scrivendo e leggendo direttamente dai registri del controllore. Il secondo utente può disabilitare le interruzioni, in modo che la seconda routine degli operatori non vada in esecuzione.

Abbiamo bisogno di un meccanismo aggiuntivo nel processore, in modo che questo non obbedisca sempre ciecamente alle istruzioni del programma corrente. Paragonando un programma ad un *testo* che il processore legge (ed esegue), aggiungiamo la possibilità di avere anche un *contesto*. Ad ogni istante ci sarà un contesto corrente, che determina come le istruzioni del programma corrente devono essere interpretate. Potremo avere un *contesto privilegiato*, nel quale verranno eseguiti i programmi degli operatori, e un contesto non privilegiato, o *utente*, in cui verranno eseguiti i programmi degli utenti. Quando il processore si trova nel contesto privilegiato obbedisce a tutte le istruzioni, come siamo abituati. Ma, quando si trova nel contesto non privilegiato, si rifiuta di eseguire le istruzioni che abilitano o disabilitano le interruzioni e tutte le istruzioni che leggono o scrivono nello spazio di I/O. Una volta aggiunto il meccanismo dei contesti, gli operatori devono fare in modo che il processore si trovi sempre nel contesto non privilegiato mentre sta eseguendo i job degli utenti. In questo modo gli utenti non possono più attuare le strategie di cui sopra.

Non dobbiamo però dimenticare che il computer è stato acquistato per servire gli utenti, non gli operatori: dobbiamo fornire all'utente 1 un modo per leggere i suoi dati dal nastro. L'utente 1 può farlo, ma non ha altra scelta se non chiamare la routine fornita dagli operatori. Dovrà però farlo utilizzando una istruzione speciale che, oltre a saltare alla routine, porta anche il processore nel contesto privilegiato, altrimenti nemmeno la routine degli operatori potrebbe interagire con il controllore del nastro. Oltre al meccanismo dei contesti, quindi, dobbiamo aggiungere al processore un meccanismo per passare da un contesto ad un altro. Questo meccanismo dovrà essere a sua volta protetto, in modo che gli utenti non possano indurre la CPU a portarsi nel contesto privilegiato e poi eseguire codice scritto da loro stessi invece che dagli operatori.

C'è un'altra cosa a cui stare attenti. Ora in memoria possiamo avere più programmi: quello scritto dagli operatori (contenente almeno le routine per i trasferimenti dal nastro) e quelli scritti dagli utenti. Nella CPU che abbiamo visto fino ad ora un programma può accedere liberamente a tutte le locazioni di memoria. Se però un utente potesse modificare le routine degli operatori, ovviamente tutto lo schema di protezione non avrebbe alcun effetto. Dobbiamo quindi anche fare in modo che, mentre è attivo il contesto non privilegiato, non sia possibile accedere alle locazioni di memoria che contengono le routine e le strutture dati degli operatori. Inoltre, ricordandoci che il computer serve agli utenti e non agli operatori, è ancora più importante garantire che il programma di un utente non possa modificare il programma di un altro utente. Se così non fosse, anche un semplice errore nel programma di un utente potrebbe causare

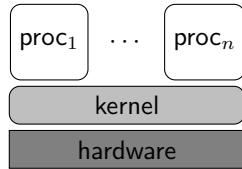


Figura 1: Rappresentazione dei livelli di privilegio e contesti.

un malfunzionamento nel programma di un altro utente, il quale poi avrebbe seri problemi a capire perché il suo programma non funziona. Possiamo evitare questo ulteriore problema prevedendo di avere tanti diversi contesti non privilegiati—per esempio, uno per ogni job. Il contesto di ogni job contiene solo il codice e i dati di quel job, ma non quelli degli altri job. Quando si passa da un job ad un altro si deve quindi cambiare anche il contesto corrente e fare in modo che, mentre è attivo un dato contesto, non sia possibile accedere alla memoria che contiene gli altri job.

Si noti che l'esempio dei sistemi batch è stato scelto perché particolarmente semplice. La necessità di poter portare avanti più programmi contemporaneamente nasce però in diversi ambiti. Ormai è data per scontata in tutti i sistemi che vanno dai supercalcolatori, ai server, ai personal computer, agli smartphone/tablet fino anche a molti sistemi embedded, con la sola eccezione dei più semplici tra questi ultimi. Il termine *job* era in uso nei sistemi batch, mentre in seguito useremo il concetto più generale di *processo*: un processo è un programma in esecuzione. 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.

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 indipendentemente dal volere degli utenti.

Riassumendo,abbiamo bisogno di:

1. un modo per definire dei contesti, privilegiati e non;
2. un modo per stabilire quale contesto è quello corrente;
3. delle regole che dicano cosa si può fare o non si può fare in ogni contesto;
4. un modo per passare da un contesto ad un altro (cambiare contesto).

Cambiare contesto per innalzare o abbassare il livello di privilegio è anche detto “cambio di livello”. Cambiare contesto per passare da un processo ad un altro è anche detto “cambio di processo”, ed è in genere una operazione molto più costosa del semplice cambio di livello.

1 La protezione nei processori Intel/AMD a 64 bit

Il meccanismo della protezione è stato aggiunto dall’Intel nel processore 80286 (1982), che era a 16 bit. Il meccanismo era strettamente intrecciato con un altro meccanismo, la segmentazione, che però non trovò molti utilizzi. Il meccanismo di protezione nell’80286 era molto sofisticato: i “livelli di privilegio” non erano soltanto due (privilegiato e non privilegiato), ma quattro; inoltre, il processore poteva eseguire interamente in hardware non solo il cambio di livello, ma anche il cambio di processo.

Con il processore 80386 (1985) l’Intel passò a 32 bit e supportò anche la paginazione, ma sempre in aggiunta alla segmentazione, estesa a 32 bit. Il meccanismo di paginazione dell’80386, però, supportava (e continua a supportare nei processori moderni) solo due livelli di privilegio: *sistema* (privilegiato) e *utente* (non privilegiato). I processori successivi (80486, Pentium, Pentium Pro, ...) non hanno cambiato questa architettura di base, ma né la segmentazione, né il meccanismo di cambio di processo via hardware sono mai stati usati in modo significativo.

Quando l’AMD ha introdotto l’architettura a 64 bit ha rimosso il cambio di processo in hardware (che possiamo dunque ignorare) e quasi completamente

disattivato il supporto alla segmentazione. Purtroppo, sempre per motivi di compatibilità, le cose sono rimaste molto più complicate di come avrebbero potuto essere. Nel seguito cercheremo di semplificare la descrizione nascondendo il più possibile i riferimenti alla segmentazione; i dettagli si potranno trovare nel codice eseguibile del sistema che realizzeremo.

1.1 Livelli di privilegio

Il processore si trova ad ogni istante o a livello (di privilegio) sistema, o a livello utente. Il livello corrente è deciso da un campo nel registro CPL.

Per quanto riguarda la protezione della memoria, facciamo subito una semplificazione (che poi rimuoveremo quando parleremo della paginazione). Immaginiamo che il processore abbia un registro, in cui si può scrivere solo da livello sistema, che contenga un indirizzo *limite* che faccia da spartiacque tra la memoria usata dal sistema e quella utilizzabile dagli utenti. Chiamiamo M1 la parte di memoria ad indirizzi inferiori al limite e M2 la rimanente. Quando il processore si trova a livello utente sono vietati tutti gli accessi alle locazioni di M1, mentre a livello sistema il limite viene ignorato (tutti gli indirizzi sono permessi). All'accensione il processore si trova a livello sistema e carica ed esegue il bootstrap loader (da una ROM). Il bootstrap loader carica le routine e le strutture dati del sistema all'inizio della memoria, quindi scrive nel registro limite l'indirizzo della prima locazione non utilizzata, definendo così la separazione tra M1 e M2.

Il cambio di livello è strettamente connesso con il meccanismo delle interruzioni. Infatti, il livello di privilegio (e dunque il contenuto di CPL) può essere cambiato solo in due modi:

- innalzato (o lasciato inalterato) passando attraverso un gate della IDT;
- abbassato (o lasciato inalterato) tramite una istruzione **iretq**.

Il termine *gate* (cancello) ci deve proprio far pensare ad un cancello che permette di entrare nel territorio privilegiato del sistema. Ci sono tre modi per attraversare un cancello. Due li conosciamo già: interruzioni esterne ed eccezioni. Il terzo lo introdurremo a breve.

Se torniamo con la mente all'esempio di partenza, vediamo che ha senso che la ricezione di una interruzione o il sollevarsi di una eccezione causino un salto ad una routine di sistema. Nell'esempio, la conclusione del trasferimento dati era segnalata da una interruzione. In risposta a questa vogliamo che vada in esecuzione una routine di sistema che faccia ripartire il job 1. Anche per le eccezioni dovrebbe essere chiaro cosa vogliamo: se un utente prova a disabilitare le interruzioni, a leggere o scrivere nei registri del controllore del nastro, o a leggere o scrivere nelle locazioni di M1, vogliamo che venga fermato. Un modo per fermarlo è che il processore, sapendo che queste operazioni sono vietate a livello utente, sollevi una "eccezione di protezione". In risposta all'eccezione vogliamo che vada in esecuzione un'altra routine del sistema, che termini il job corrente e ne metta in esecuzione un altro. Si capisce anche perché deve essere

la **iretq**, che si trova in fondo a tutte le routine di risposta alle interruzioni ed eccezioni, l'istruzione che si preoccupa di riportare il processore a livello utente.

Affinché questo meccanismo sia efficace gli utenti non devono essere in grado di modificare il contenuto della IDT. Questo si ottiene rendendo privilegiata l'istruzione **lidt** (che carica l'indirizzo della IDT nel registro IDTR che il processore usa per accedere alla tabella) e allocando la IDT nella memoria M1. Questo può essere fatto in fase di inizializzazione del sistema: la IDT è semplicemente una delle strutture dati del sistema, caricate in M1 dal bootstrap loader. Dopo il caricamento il bootstrap loader salta ad una ben precisa routine del sistema appena caricato, la quale si preoccupa di inizializzare tutte le strutture dati di sistema (compresa la IDT) e il processore, in particolare caricando il registro IDTR con l'indirizzo della IDT.

Ora, se torniamo ancora una volta all'esempio, ci rendiamo conto che abbiamo bisogno di un terzo modo per attraversare i cancelli: l'utente 1 deve poter invocare una routine di sistema quando ha bisogno di caricare i dati dal nastro. Per far questo il processore prevede una nuova istruzione:

```
int $tipo
```

Questa istruzione ha un unico parametro immediato, *tipo*, che deve essere un numero tra 0 e 255. Il tipo ha lo stesso significato del tipo delle interruzioni esterne e delle eccezioni: è utilizzato per accedere ad un gate della IDT, dove il processore trova l'indirizzo della routine a cui saltare e l'informazione che gli dice se il livello di privilegio deve essere innalzato. Si dice che l'istruzione genera una “interruzione software”. Le interruzioni software sono del tutto analoghe a quelle esterne ed alle eccezioni di tipo trap, con salvataggio in pila di informazioni analoghe. In particolare, l'instruction pointer salvato in pila è quello dell'istruzione successiva alla **int**. Anche le routine di risposta alle interruzioni software devono terminare con **iretq** e non **ret**.

Le routine che vanno in esecuzione tramite una **int** prendono comunemente il nome di *primitive di sistema*. Chi scrive il codice di sistema deve fornire ai suoi utenti la necessaria documentazione su: quali sono le primitive disponibili; quale tipo è necessario usare per invocare ciascuna primitiva; quali parametri ciascuna primitiva si aspetta e come le vanno passati (tipicamente tramite i registri). L'istruzione **int** svolge un compito simile a quello di una **call**. Ci si può chiedere come mai serva una nuova istruzione per invocare una primitiva, invece di aggiungere la possibilità di innalzare il livello di privilegio alla normale **call**. Ma l'istruzione **call** specifica direttamente l'indirizzo a cui saltare, e questo causa due problemi:

1. l'utente potrebbe saltare in mezzo al codice delle primitive (per esempio, per scavalcare dei controlli);
2. se gli indirizzi vengono specificati in forma simbolica, i programmi utente dovrebbero essere collegati con il codice di sistema per poter risolvere i simboli.

Il primo problema è di gran lunga più grave, mentre il secondo è solo una scomodità. L'istruzione **int** non ha questi problemi, in quanto l'utente specifica solo il tipo della primitiva che intende invocare, mentre l'indirizzo a cui saltare è scritto nella IDT, a cui lui non ha accesso. C'è un ultimo vantaggio, che per il momento non possiamo apprezzare pienamente: è molto comodo, per chi scrive il sistema, che i modi con cui si può accedere al sistema siano il più possibile uniformi.

1.2 Interruzioni e protezione

Vediamo più in dettaglio come funziona il meccanismo delle interruzioni. Ogni gate della IDT occupa 16 byte e contiene le seguenti informazioni:

- un bit P (Presenza), che indica se il gate contiene informazioni significative (il sistema non deve necessariamente utilizzare tutti i gate);
- il puntatore alla routine a cui saltare (8 byte);
- un bit I/T che indica se il gate è di Interrupt o Trap;
- un campo L che indica il livello di privilegio a cui il processore si deve portare dopo aver attraversato il gate;
- un campo DPL (Descriptor Privilege Level, chiamato anche GL per Gate Level) che indica il livello di privilegio minimo che il processore deve avere per poter attraversare il gate nel caso di interruzione software.

Quando il processore accetta una interruzione esterna, genera una eccezione o esegue una interruzione software,

1. si procura il *tipo* dell'interruzione:
 - in caso di eccezione, il tipo è implicito;
 - in caso di interruzione esterna, riceve il tipo dall'APIC;
 - in caso di interruzione software, il tipo è il parametro dell'istruzione **int**.

Usa quindi il tipo come indice nella tabella IDT, per accedere al corrispondente gate.

2. Se il bit P del gate è zero, il processore smette di gestire l'interruzione e genera una eccezione per “gate non presente” (tipo 11).
3. Altrimenti, se sta gestendo una interruzione software (o eseguendo una **int3**), confronta il livello corrente (registro CPL) con il campo DPL del gate. Se il livello corrente è meno privilegiato di DPL, genera una eccezione di protezione (tipo 13).
4. Altrimenti, confronta il CPL con il campo L. Se L è inferiore, genera una eccezione di protezione (tipo 13).

5. Altrimenti, salva in un registro di appoggio (chiamiamolo SRSP) il contenuto corrente di **rsp**.
6. Se CPL è diverso da L, esegue un *cambio di pila*, caricando un nuovo valore in **rsp** (si veda la Sezione 1.3 più avanti per sapere da dove viene prelevato il nuovo valore).
7. Salva in pila (la nuova o la vecchia, a seconda che al punto precedente abbia cambiato pila o meno) 5 parole lunghe (8 byte ciascuna), in ordine:
 - una parola lunga non significativa (segmentazione ...);
 - il contenuto di SRSP (questo punta alla vecchia pila, nel caso di cambio pila);
 - il contenuto di **rflags**;
 - il contenuto di CPL;
 - il contenuto di **rip**.
 (in cima alla pila c'è dunque **rip**).
8. Azzera in ogni caso TF (disabilita una eventuale modalità Single Step) e azzera IF (Interrupt Flag) solo se il gate è di tipo Interrupt (campo I/T del gate).
9. Salta infine all'indirizzo della routine puntata dal gate.

Il controllo eseguito al punto 3 può essere usato per imporre che l'utente usi l'istruzione **int** solo con i tipi associati a primitive di sistema, e non con i tipi associati alle interruzioni esterne o alle eccezioni. I programmati di sistema possono impostare DPL=sistema in tutti i gate delle interruzioni esterne e delle eccezioni, e DPL=utente in quelli che puntano alle primitive. In questo modo le interruzioni esterne e le eccezioni vengono gestite normalmente (perché DPL non viene controllato in questi casi), mentre ogni tentativo dell'utente di usare un gate che non è assegnato ad una primitiva genera una eccezione di protezione.

Il controllo effettuato al punto 4 corrisponde a quanto si era detto: le interruzioni devono solo poter innalzare il livello di privilegio (o lasciarlo inalterato) e mai abbassarlo. Il punto è che una richiesta di interruzione manda in esecuzione una routine e, normalmente, ci aspettiamo che questa routine faccia ciò che deve fare e poi ritorni al programma principale. Ma se la routine è di livello utente non possiamo fare nessuna assunzione, nemmeno sul fatto che prima o poi termini. Quindi, se il processore si trova a livello sistema (e dunque sta svolgendo operazioni importanti), è pericoloso che venga interrotto da una routine di livello utente. È comunque abbastanza insolito che un gate di interruzione contenga L=utente e noi assumeremo che non avvenga, rendendo dunque ridondante questo controllo.

Si noti che il meccanismo contempla anche i casi CPL=utente/L=utente e CPL=sistema/L=sistema. Possiamo ignorare quello con L=utente, per quando

appena detto. Il caso CPL=sistema/L=sistema si può invece facilmente verificare: se il sistema invoca esso stesso una primitiva, o se una routine di sistema di tipo trap viene interrotta da una eccezione o da una interruzione esterna. In questi casi non ci sarà cambio pila.

Il cambio pila eseguito al punto 6 in caso di innalzamento del livello di privilegio (da utente a sistema) ha due motivazioni:

- il processore deve garantire di poter scrivere le 5 parole lunghe senza sovrascrivere altre cose, e non può dunque fidarsi del contenuto di **rsp** controllato dall'utente;
- è bene che queste informazioni siano salvate nella memoria di sistema (M1), in modo che l'utente non le possa corrompere. In particolare, è bene che l'utente non possa modificare il valore salvato di CPL, che dice a che livello si trovava il processore prima dell'interruzione.

Si noti però che la seconda motivazione è importante solo quando il sistema dispone di più di un processore (cosa che noi non prenderemo in considerazione). In presenza di un unico processore, infatti, il codice di sistema stesso potrebbe copiare le informazioni dalla pila in memoria M1, senza il timore che il codice utente possa interferire nella copia. La prima motivazione, invece, resta vera in ogni caso.

L'istruzione **iretq** svolge le operazioni opposte a quelle del meccanismo di interruzione.

(a) Confronta il valore corrente di CPL con quello salvato in pila; se quello salvato è più privilegiato di quello corrente genera una eccezione di protezione (tipo 13).

(b) Ripristina i valori di **rip**, CPL, **rflags** e **rsp** leggendo i corrispondenti valori dalla pila.

Si noti che con il ripristino di **rsp** processore ritorna ad usare la pila originaria, nel caso questa fosse stata cambiata all'avvio dell'interruzione. Il ripristino di **rflags** ripristina in particolare i vecchi valori di TF e IF, eventualmente riabilitando il Single Step e le interruzioni esterne. Il ripristino di CPL riporta il processore al livello di privilegio precedente.

Il controllo eseguito al punto (a) corrisponde a quanto detto in precedenza: l'istruzione **iretq** può solo abbassare (o lasciare inalterato) il livello di privilegio, mai innalzarlo. Diversamente dal complementare controllo nel caso delle interruzioni (punto 4), questo controllo è molto importante. Se non venisse effettuato sarebbe facile per l'utente eseguire codice di suo piacimento a livello sistema (come?).

Si noti che la routine di inizializzazione del sistema deve avere un modo per poter passare a livello utente una volta che tutte le strutture dati sono state inizializzate. Per farlo è sufficiente che prepari una pila nello stesso stato in cui l'avrebbe lasciata una interruzione proveniente dal livello utente, ed esegua una **iretq**. Da quel momento in poi il controllo passa agli utenti. Il sistema

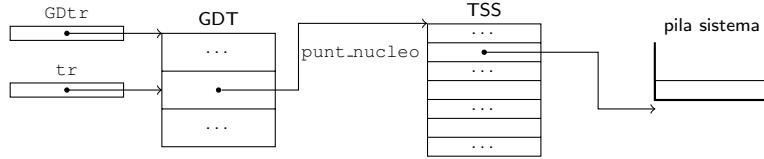


Figura 2: Strutture dati del sistema associate ad ogni processo.

interviene solo in risposta alle richieste di interruzione esterne, le eccezioni, e le interruzioni software.

1.3 Il registro **tr** e i Task State Segment

Al punto 6 della Sezione precedente, abbiamo visto che il processore cambia pila quando deve innalzare il livello di privilegio. In puntatore alla nuova pila è prelevato dal *Task State Segment* (TSS) corrente. I TSS sono delle strutture in memoria contenenti spazio per diversi campi, tra cui il puntatore alla pila che ci interessa. Erano usati dal meccanismo di cambio di processo in hardware, che, come abbiamo visto, non è più disponibile. È necessario, però, continuare a definirne almeno uno, per compatibilità con il fondamentale meccanismo delle interruzioni. Chiameremo *punt_nucleo* il campo del TSS che punta alla pila di livello sistema.

Nell'idea originaria ci doveva essere un TSS per ogni processo ma, ad ogni istante, uno solo è quello “attivo”. Il registro TR (Task Register) serve a identificare il TSS attivo e può essere caricato usando l'istruzione **ltr**. Tutti i TSS esistenti sono descritti da una tabella GDT (Global Descriptor Table), puntata dal registro GTDR che può essere caricato con l'istruzione **lgdt**. La GDT contiene alcune entrate che non ci interessano (segmentazione...) e una entrata per ogni TSS. Le entrate relative ai TSS contengono l'indirizzo di partenza (base) del TSS e la sua grandezza in byte. Il registro TR contiene l'offset (rispetto all'inizio della GDT) dell'entrata della GDT che punta al TSS corrente. Si veda la Figura 2.

Ovviamente, le istruzioni **lgdt** e **ltr** sono privilegiate e la GDT e tutti i TSS devono trovarsi in M1.

Nell'idea originaria, il software di sistema deve aggiornare TR ogni volta che cambia processo. Noi useremo un unico TSS, caricando TR una sola volta in fase di inizializzazione. Avremo però una pila sistema diversa per ogni processo e, ad ogni cambio di processo, dovremo aggiornare il campo *punt_nucleo* di quell'unico TSS.

1.4 Livelli di privilegio e operazioni di I/O

Siamo partiti da un esempio in cui volevamo vietare agli utenti di abilitare/-disabilitare le interruzioni e accedere allo spazio di I/O. Nell'architettura Intel queste operazioni non sono automaticamente vietate quando il processore si

trova a livello utente. La possibilità o meno di eseguire queste operazioni è determinata, invece, dal campo IOPL (I/O Privilege Level) che si trova nel registro dei flag. Questo campo specifica il livello di privilegio minimo che il processore deve avere per poter abilitare o disabilitare le interruzioni ed eseguire le istruzioni `in` e `out` (in tutte le loro varianti). Il campo IOPL può essere modificato solo a livello sistema. Il tentativo di modificarli da livello utente (per esempio tramite **`pushf/popf`**) non genera, come ci potremmo aspettare, una eccezione di protezione, ma viene semplicemente ignorato: IOPL continua a mantenere il suo valore senza che venga segnalato alcun errore. Lo stesso accade se si tenta di modificare il flag IF tramite **`popf`**.

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 dati, 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 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’altro¹.
- 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).

¹Nel 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 quali 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 interpretare 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 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.

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 strutture 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 bootstrap 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*, contenenti alcune funzioni di utilità, e *utente.s*, contenente 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 programmi di utilità lanciare il comando *compile*, già usato per gli esempi di I/O²

²Nel 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 *bootstrap* 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 modalità 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 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 libce³. 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

³I 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 - ... azzerrati 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, 0000080000000000)
15 INF - sis/priv [0000080000000000, 0000100000000000)
16 INF - io /cond [0000010000000000, 0000180000000000)
17 INF - usr/cond [ffff800000000000, ffff000000000000)
18 INF - usr/priv [ffffc0000000000, 0000000000000000)
19 INF - Crea finestra sulla memoria centrale: [0000000000001000, 0000000002000000)
20 INF - Crea finestra per memory-mapped-I/O: [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 [ffff800000020000, ffff800000004000)
29 INF - - heap: [ffff80000004000, ffff8000000104000)
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 - Crea il processo main_sistema (id = 0)
36 INF - Crea 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 Crea 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 Crea 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 eseguire 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=000000000000202 [- - - - IF - - - - - - -, IOPL=SISTEMA]
11 WRN 5 RAX=00000000000b8008 RBX=ffff800000002fe0 RCX=0000000000000000 RDX=ffff800000004000
12 WRN 5 RDI=ffff800000004000 RSI=0000000000100000 RBP=fffffffffffe0 RSP=fffffffffffe0
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 restituisce 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.

Realizzazione dei processi

G. Lettieri

29 Aprile 2021

1 I processi

Vediamo ora come i processi sono realizzati in generale, e poi come sono stati implementati nel sistema didattico.

1.1 Stati di esecuzione dei processi

Durante la sua vita un processo si trova in uno degli *stati di esecuzione* illustrati in Figura 

I processi devono essere prima di tutto “attivati”, in modo che possano cominciare ad essere eseguiti. L’attivazione comporta la creazione di tutte le strutture dati necessarie (descrittore di processo, pile, etc.). In alcuni sistemi i processi da attivare sono decisi staticamente all’avvio del sistema. Noi realizzeremo il caso in cui i processi possono essere creati dinamicamente da altri processi (tranne ovviamente il primo processo, che sarà creato dal sistema stesso all’avvio).

Se un processo si trova in “esecuzione”, il processore sta eseguendo le sue istruzioni: il processo ha il controllo del processore e lo stato del processo (contenuto dei registri e della memoria) cambia nel tempo. Se abbiamo un solo processore (come stiamo supponendo in tutto il corso), un solo processo per volta può trovarsi in esecuzione. In tutti gli altri stati di esecuzione lo stato del processo resta costante nel tempo.

Mentre si trova in esecuzione un processo può chiedere di terminare, oppure di sospendersi in attesa di un evento. Esempi di evento sono il completamento di una operazione di I/O, o il passaggio di un determinato intervallo di tempo o anche, come vedremo, l’arrivo di un generico “segnale” da parte di un altro processo. Quando l’evento atteso si verifica il processo diventa “pronto” (può anche accadere che vada direttamente in esecuzione, anche se ciò non è mostrato esplicitamente in figura).

I processi che si trovano nello stato (di esecuzione) “pronto” sono processi che potrebbero proseguire se avessimo a disposizione sufficienti processori: sono

¹ Attenzione a non confondere lo stato *di esecuzione* di un processo (che è uno tra nuovo, pronto, esecuzione, bloccato e terminato) con lo *stato del processo*, di cui abbiamo parlato precedentemente, e che consiste nel contenuto dei registri e della memoria in un qualche punto della vita del processo.

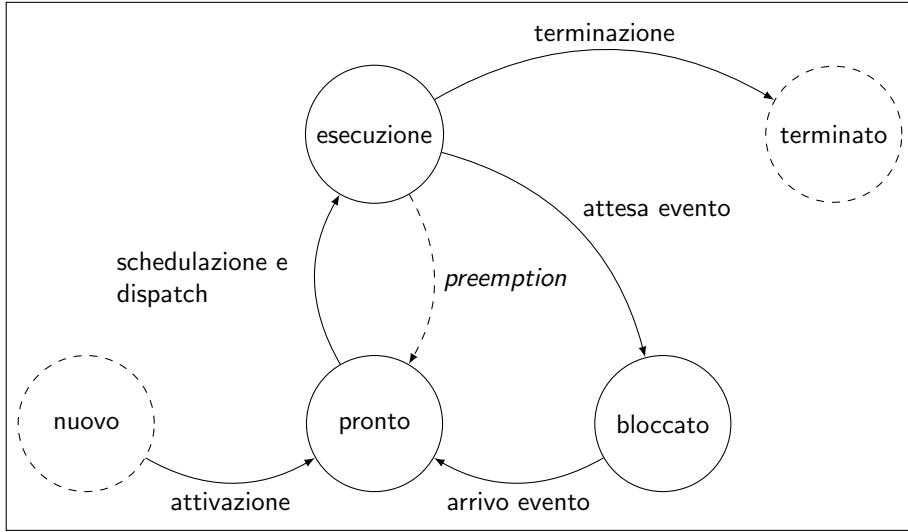


Figura 1: Stati di esecuzione un processo.

fermi solo perché il processore è già impegnato a portare avanti un altro processo. Un processo pronto può essere scelto per andare in esecuzione tramite una operazione che è detta di “*schedulazione*”. Il processo andrà effettivamente in esecuzione (prendendo il controllo del processore) tramite una successiva operazione che è detta di *dispatch*. In alcuni sistemi (ma non in quello che realizzeremo) schedulazione e dispatch sono combinate in un’unica azione. In ogni caso l’azione di dispatch segue dopo poco quella di schedulazione, e il fatto che siano distinte è solo un dettaglio implementativo.

Ovviamente, al momento del dispatch, il processo che era precedentemente in esecuzione deve aver prima liberato il processore, per esempio chiedendo di bloccarsi e passando nello stato “*bloccato*”. Un processo in esecuzione può anche essere costretto a liberare forzatamente il processore, con una azione che è detta di “*preemption*” (prelazione). La preemption può avvenire sia in seguito ad una interruzione o eccezione, sia come effetto collaterale di una azione richiesta dal processo in esecuzione stesso (vedremo che nel nostro caso ciò può accadere quando un processo invia un segnale ad un altro che lo stava aspettando). Il caso di preemption differisce dal caso di blocco, in quanto il processo passa nello stato “*pronto*” e non nello stato “*bloccato*”: il processo non sta attendendo un evento e non è più in esecuzione soltanto perché un altro processo sta occupando il processore.

Si noti che nei sistemi senza preemption un processo può occupare il processore indefinitamente (per esempio, eseguendo un ciclo infinito) senza lasciare mai il processore agli altri processi.

Sono possibili tante strategie di schedulazione. Nella strategia *time-sharing* (a divisione di tempo), per esempio, il processore viene assegnato ad ogni pro-

cesso pronto per un tempo massimo, passato il quale un timer interrompe il processore causando una preemption con schedulazione e dispatch del prossimo processo pronto. Noi realizzeremo un sistema a *priorità fissa*: ad ogni processo è assegnata un priorità numerica al momento della creazione e il sistema si impegna a garantire che, ad ogni istante, si trovi in esecuzione il processo che ha la massima priorità tra tutti quelli pronti. Questo ci permette di dover eseguire una azione di schedulazione solo quando un processo passa da “esecuzione” a “bloccato” oppure da “bloccato” a “pronto”. Nel primo caso il processore si libera, e dunque dobbiamo mettere in esecuzione il processo a maggiore priorità tra i pronti. Nel secondo caso c’è un novo processo pronto, che potrebbe avere priorità maggiore di quello attualmente in esecuzione: per rispettare la regola che abbiamo promesso di garantire potremmo fare preemption sul processo in esecuzione. Si noti che anche quando un processo P_1 ne attiva un altro P_2 ci troviamo in una situazione simile: abbiamo un nuovo processo pronto (P_2) mentre un altro (P_1) è in esecuzione. Noi però garantiremo che i processi non possano attivarne altri a priorità maggiore della propria, quindi non sarà mai necessaria una preemption in questo caso.

1.2 Realizzazione dei processi (nel sistema didattico)

Il nostro sistema, per semplicità, sarà mono-utente e mono-programma (perché prevediamo un singolo modulo `utente`), ma sarà comunque multiprocesso, nel senso che il programma potrà creare tanti processi a cui far eseguire funzioni definite nel programma stesso. I sistemi multi-processo si distinguono comunemente in “sistemi a memoria comune”, in cui tutti i processi hanno accesso alla stessa memoria, e “sistemi a scambio di messaggi”, in cui ogni processo ha una memoria completamente privata (e può comunicare con gli altri processi solo tramite messaggi). Noi realizzeremo un modello ibrido, perché questo ci permetterà di esplorare più dettagliatamente il meccanismo che rende possibile condividere o non condividere la memoria tra i processi (cioé la paginazione, come vedremo in seguito). Nel nostro sistema ogni processo utente ha una sua pila utente distinta da quella di tutti gli altri processi, a cui ha accesso esclusivo, mentre le sezioni `.text`, `.data` e `.bss`, e anche lo heap, sono condivise tra tutti. In pratica sono condivise tutte le entità globali definite nel modulo utente, più lo heap.

Dal punto di vista del sistema ogni processo ha inoltre un proprio descrittore di processo e una propria pila sistema. Lo scopo del descrittore di processo è di contenere tutte le informazioni che il sistema deve ricordare relativamente al processo. In particolare, il descrittore contiene un campo `contesto` destinato a contenere l’ultima “fotografia” scattata sullo stato del processore (vale a dire, il contenuto di tutti i registri del processore). Si ricordi che una foto completa dello stato del processo deve contenere anche lo stato di tutta la memoria. Per il momento supponiamo che lo stato di tutta la memoria del processo sia conservato su un hard disk e che nel descrittore di processo ci saranno le informazioni necessarie a recuperare tale stato dall’hard disk (almeno per la parte che non è a comune con gli altri processi).

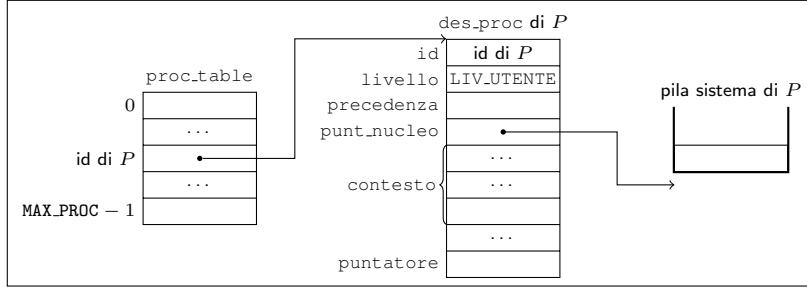


Figura 2: Strutture dati del sistema associate ad ogni processo utente.

Per gestire gli stati di Figura 1 faremo ricorso a *code di processi* realizzate come liste di descrittori di processo. La Figura 2 mostra le strutture dati che il modulo sistema deve associare ad ogni processo (con l'esclusione delle strutture dati relative alla memoria del processo). Per il descrittore di processo definiamo una struttura `des_proc` che memorizza l'identificatore numerico del processo (campo `id`), il suo livello di privilegio (utente o sistema) e la sua precedenza, come decisi al momento della sua creazione (campi `livello` e `precedenza`). Il campo `contesto` memorizza la copia privata dei registri del processore, che fanno parte appunto del contesto del processo. Questo campo è un array di `natq` (interi senza segno a 64 bit). Ogni registro ha un posto assegnato all'interno di questo array: definiremo delle costanti (`I_RAX`, `I_RBX`, etc.) per evitare di ricordare gli indici numerici. Osserviamo anche il campo `punt_nucleo`, che punta alla base della pila sistema del processo, e il campo `puntatore`, che serve a realizzare le code di processi a cui abbiamo accennato poco sopra.

Ricordiamo che il meccanismo di interruzione, nel caso in cui debba cambiare la pila corrente, prende l'indirizzo della nuova pila dal segmento TSS identificato dal registro TR. Noi allocheremo un unico segmento TSS e inizializzeremo il registro TR una sola volta all'avvio del sistema. Poi, per fare in modo che il meccanismo delle interruzioni utilizzi la pila sistema del processo corrente, dovranno sovrascrivere opportunamente il segmento TSS ogni volta che cambiamo processo.

Il processo attualmente in esecuzione sarà identificato dalla variabile globale `esecuzione`, di tipo puntatore a `des_proc`. I processi pronti si troveranno in una coda globale, la cui testa sarà puntata dalla variable `pronti`, anch'essa di tipo puntatore a `des_proc`. Predisporremo inoltre una coda diversa per ogni tipo di evento atteso dai processi bloccati. Manterremo tutte queste code ordinate in base al campo priorità. In questo modo, in particolare, la schedulazione di un processo pronto può essere eseguita semplicemente estraendo il `des_proc` in testa alla coda `pronti`.

Per evitare di dover gestire in modo speciale il caso in cui tutti i processi sono bloccati è sufficiente che il sistema crei un processo a priorità minima che sia sempre pronto, detto processo dummy. Tale processo può eseguire un ciclo infinito. Nel nostro caso il processo dummy controllerà ciclicamente il numero

di processi attualmente esistenti nel sistema. Quando scopre che tutti gli altri processi sono terminati, il processo dummy può eseguire lo *shutdown* del sistema.

1.3 Cambio di processo

Il cambio di processo può avvenire solo quando il processo in esecuzione si porta a livello sistema, cosa che può avvenire solo nei seguenti modi:

- se il processo stesso esegue una istruzione **int**;
- se il processore genera una eccezione (per es., page fault);
- se il processore accetta una interruzione esterna.

In tutti e tre i casi il processore esegue azioni simili: consulta una entrata (“cancello”) della tabella IDT per prelevare l’indirizzo a cui saltare, salvando in pila l’indirizzo a cui ritornare. In base ai flag contenuti nel cancello, il processore può eseguire anche altre azioni: innalzare il livello di privilegio del processore (in base al valore del campo L); disabilitare le interruzioni (in base al valore del campo I/T). Se deve innalzare il livello di privilegio, provvede anche a cambiare pila (prelevando il puntatore alla nuova pila campo `punt_nucleo` del descrittore di processo puntato dal registro TR). Tutti i cancelli del nostro sistema prevedono l’innalzamento del livello di privilegio, quindi in tutti e tre i casi il processore passerà ad usare la pila sistema del processo corrente. In questa pila salverà il puntatore alla vecchia pila, il contenuto del registro **rflags**, il livello di privilegio precedente l’innalzamento e l’indirizzo della prossima istruzione da eseguire. Vedremo anche che tutti i cancelli che portano al modulo sistema prevedono la disabilitazione delle interruzioni (il campo I/T deve dunque specificare il tipo Interrupt e non Trap).

Inoltre, predisporremo le cose in modo che il codice a cui si salta in tutti e tre i casi segua il seguente schema:

```
call salva_stato  
...  
call carica_stato  
iretq
```

L’ingresso nel modulo sistema, subito dopo le operazioni svolte dal meccanismo di interruzione, passa quindi per la funzione `salva_stato`. L’uscita dal modulo sistema (con successivo ritorno al modo utente tramite `iretq`) passa invece dalla funzione `carica_stato`. La funzione `salva_stato` salva lo stato del processore nel descrittore del processo identificato dalla variabile `esecuzione`, mentre la seconda carica nel processore lo stato contenuto nel descrittore del processo identificato da `esecuzione`. Vedremo in seguito che in alcuni casi le chiamate a `salva_stato` e `carica_stato` si possono, oppure *devono*, essere omesse, ma per il momento possiamo assicurare che siano sempre presenti.

In Figura 3 mostriamo un esempio con due processi, P_1 , P_2 (oltre al processo dummy, sempre presente) nel momento in cui P_1 , che è in esecuzione, si porta

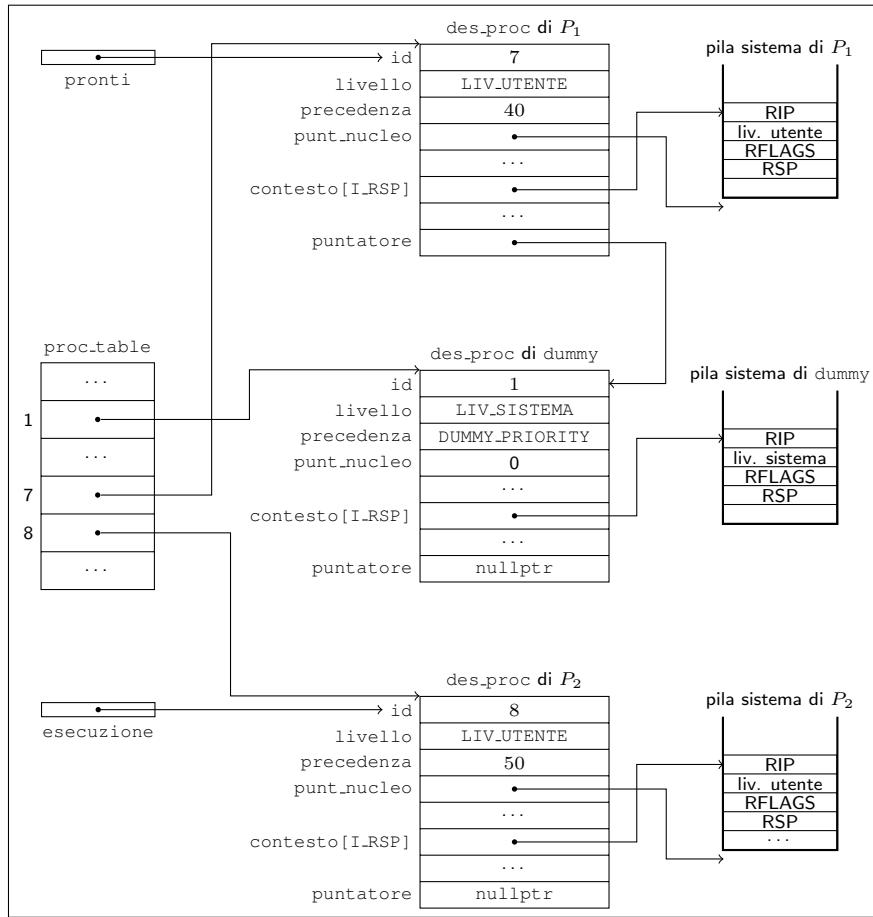


Figura 3: Esempio di istanziazione delle strutture dati per la gestione dei processi.

a livello sistema per uno qualunque dei tre motivi di cui sopra, mentre P_2 era pronto. La Figura mostra lo stato delle strutture dati dopo il ritorno dalla **call** `salva_stato`.

La “fotografia” dello stato di P_1 e P_2 è costituita in parte dalle informazioni salvate in pila sistema dal meccanismo di interruzione (in particolare, il contenuto dei registri **rip** e **rsp**) e in parte dal contenuto del descrittore di processo come aggiornato dalla `salva_stato`.

Attenzione ai due valori salvati di **rsp**: quello che fa parte dello stato del processo è quello che punta alla pila utente e che si trova salvato in pila sistema. La `salva_stato` salva anche (nel campo `contesto[I_RSP]` del descrittore di processo) il contenuto del registro **rsp** come lasciato dal processore dopo il cambio di pila e il successivo salvataggio in questa delle cinque parole lunghe. La funzione `carica_stato` ripristinerà anche questo registro in modo che l’istruzione **iretq** finale possa estrarre dalla pila sistema queste informazioni, facendo dunque proseguire il processo dal punto in cui era stato interrotto.

Si noti che, quando il processore sta eseguendo codice del modulo sistema, *tutti* i processi si trovano in uno stato simile: non solo quello in esecuzione, ma anche tutti i pronti e tutti quelli bloccati. Infatti, se questi erano stati precedentemente in esecuzione, l’unico modo in cui possono esserne usciti è passando dal meccanismo delle interruzioni, in uno dei tre modi possibili. Tutti saranno dunque passati anche dalla `salva_stato`, e dunque i loro descrittori di processo e pile sistema saranno in una configurazione adatta ad essere interpretati da una `carica_stato` seguita da una **iretq**². In Figura 3 questo è illustrato dal fatto che tutte le pile sistema contengono lo stesso tipo di informazioni. Per passare da un processo ad un altro è dunque sufficiente cambiare il valore della variable `esecuzione` in un momento qualunque tra la **call** `salva_stato` e la **call** `carica_stato`. La `carica_stato` finale caricherà (insieme agli altri registri) il valore di **rsp** che punta alla pila sistema del nuovo processo, e la successiva **iretq** farà ripartire quest’ultimo. In questo modo possiamo entrare nel sistema eseguendo un processo e, al ritorno, saltare ad un altro.

1.4 Attivazione di un processo

Nel nostro sistema un processo ne può attivare un altro invocando la primitiva `activate_p()`. La Figura 4 mostra un esempio di programma utente che usa la primitiva per attivare un nuovo processo. La primitiva accetta i seguenti parametri:

- un puntatore `f` alla funzione che il nuovo processo dovrà eseguire; la funzione deve essere di tipo **void (int)**, cioè accettare un unico parametro, di tipo **int**, e non restituire niente (`mioproc` in Figura);
- un’espressione di `natq`, che sarà il valore ricevuto come argomento dalla funzione eseguita dal processo (10 in Figura);

²Per i processi che non erano ancora mai andati in esecuzione si veda la sezione successiva.

```

1 #include <all.h>
2
3 void mioproc(natq i)
4 {
5     printf("mioproc: i = %d", i);
6     pause();
7     terminate_p();
8 }
9
10 int main()
11 {
12     natl id = activate_p(mioproc, 10, 20, LIV_UTENTE);
13     printf("Ho creato il processo %d", id);
14     terminate_p();
15 }
```

Figura 4: Un esempio di programma utente (file `utente/utente.cpp`).

- un'espressione di tipo `natl` che indica la priorità del processo (20 in Figura);
- uno tra `LIV_UTENTE` o `LIV_SISTEMA`, per indicare se il nuovo processo deve essere di tipo utente o sistema.

Si noti che un processo utente non può creare un processo di livello sistema e non può creare un processo a priorità maggiore della propria. La primitiva restituisce l'identificatore numerico del nuovo processo, o `0xFFFFFFFF` in caso di errore.

Nell'esempio di Figura 4 il nuovo processo eseguirà `mioproc(10)`. Si noti che nulla vieta di invocare `activate_p()` più volte, anche usando lo stesso puntatore a funzione. Ogni invocazione attiverà un nuovo processo (si ricordi la distinzione tra processo e programma: `mioproc` è solo il programma).

1.5 Realizzazione della `activate_p()`

La primitiva `activate_p()` dovrà fare in modo che il nuovo processo, quando verrà messo in esecuzione la prima volta, parta dalla prima istruzione della funzione `f`, usando una nuova pila utente.

Per attivare un processo è necessario allocare e inizializzare tutte le sue strutture dati: descrittore di processo (`des_proc`) e pila sistema. Per capire come inizializzarle, pensiamo a cosa accadrà al processo dopo averlo attivato: sarà inserito in coda pronti (si veda la Figura 1), dove prima o poi verrà schedulato e portato in esecuzione. Sappiamo che ciò avverrà facendo puntare `esecuzione` al nuovo `des_proc`, quindi eseguendo `call carica_stato` e infine `iretq`. Per i processi che erano arrivati in coda pronti essendo stati precedentemente in esecuzione, il descrittore letto dalla `carica_stato` e la pila sistema letta dalla `iretq` erano stati opportunamente inizializzati (dalla `salva_stato` e dal meccanismo di interruzione) l'ultima volta che il processo era uscito dallo stato esecuzione. Se vogliamo che il nuovo processo si comporti come tutti gli altri che sono già in coda pronti, possiamo inizializzare il descrittore di processo e

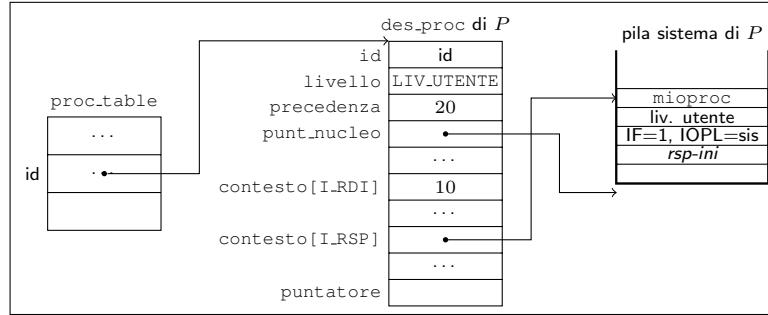


Figura 5: Attivazione del processo utente di Figura 4. Tutte le strutture dati mostrate si trovano nella parte M1 della memoria.

pila sistema *come se* anch'esso si fosse precedentemente portato da livello utente a livello sistema, subito prima di eseguire la sua prima istruzione (cioè quella all'indirizzo `mioproc`, nell'esempio di Figura 4). La Figura 5 mostra il risultato finale. In pila sistema scriviamo l'indirizzo di partenza (`mioproc` nell'esempio), il codice del livello utente, un campo `rflags` cof flag `IF=1` e `IOPL=sistema`, e `rsp-init`, che punta alla pila utente opportunamente allocata. Inizializziamo il campo `contesto[I_RSP]` in modo che, in seguito alla `carica_stato`, il registro `rsp` del processore punti alle informazioni che abbiamo scritto in pila sistema. In questo modo la `iretq` che segue sempre la `call carica_stato` farà saltare il processore all'istruzione di indirizzo `mioproc`, a livello utente, con le interruzioni abilitate (e l'impossibilità di disattivarle) e pronto a usare la pila utente puntata da `rsp-init`.

Si noti che il campo `punt_nucleo` deve puntare comunque alla base della pila sistema come se questa fosse vuota. Questo campo, infatti, verrà utilizzato dal meccanismo delle interruzioni quando il processo sarà ormai in esecuzione a livello utente. Quando un processo si trova a livello utente la sua pila sistema è sempre vuota: si riempie passando da utente a sistema e si svuota al ritorno.

Si noti che per fare in modo che la funzione che il processo eseguirà riceva il parametro che l'utente ha chiesto (secondo argomento della `activate_p()`) dobbiamo fare in modo che questo si trovi nel registro `rdi`. Per ottenere questo è sufficiente che all'attivazione del processo scriviamo il parametro attuale nel campo `contesto[I_RDI]`. La prima `carica_stato` lo porterà poi nel registro `rdi`.

Per completare l'attivazione dobbiamo anche occupare una nuova entrata della `proc_table`, scrivendovi dentro il puntatore al nuovo `des_proc`. L'indice di questa entrata nella `proc_table` funge anche da identificatore del processo. Scriviamo l'identificatore nel campo `id` del nuovo `des_proc`. Il campo `precedenza` del `des_proc` deve contenere la priorità (`prio`) del nuovo processo, che è semplicemente quella ricevuta come terzo parametro dalla `activate_p()`.

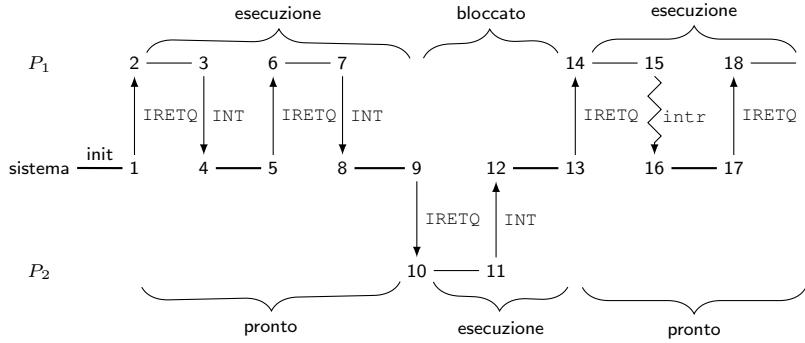


Figura 6: Un esempio di evoluzione del sistema con due processi.

1.6 Esempio

La Figura 6 mostra un esempio di una possibile evoluzione del sistema con due processi, P_1 e P_2 , con P_1 che ha una priorità maggiore di P_2 . Durante l'inizializzazione supponiamo che vengano attivati sia P_1 che P_2 e inseriti in coda pronti. All'istante 1 il sistema esegue la **iretq** che salta alla prima istruzione di P_1 . All'istante 3 P_1 invoca una primitiva del sistema tramite una istruzione **int**. Subito dopo, all'istante 4, il sistema salva lo stato di P_1 nel suo descrittore di processo. Una volta terminata, la primitiva torna al processo P_1 senza cambiare processo, all'istante 5. Si noti che il descrittore di processo viene aggiornato solo quando un processo si porta da livello utente a sistema. Nel caso di P_1 ciò avviene agli istanti 4, 8, e 16. Per tutto il resto del tempo il descrittore di processo continua a memorizzare l'ultimo stato salvato. Inoltre, si faccia attenzione a *quale* stato viene salvato: all'istante 4 viene salvato lo stato che permette a P_1 di ripartire dal punto 6, dove ci sarà la prima istruzione successiva alla **int** che P_1 aveva eseguito all'istante 3. Ciò è semplice ed intuitivo da ricordare quando il sistema *non* cambia processo tra la **salva_stato** e la **carica_stato**. Ma si ricordi che ciò avviene sempre: all'istante 7 vediamo P_1 invocare di nuovo una primitiva, portandosi a livello sistema (istante 8). La situazione delle strutture dati del sistema è ora quella descritta in Figura 3. Il RIP salvato nella pila sistema di P_1 è quello che punta all'istruzione successiva alla **int** appena eseguita. Questa volta supponiamo che la primitiva blocchi P_1 e scheduli P_2 : la pila sistema attiva (cioè, puntata dal registro RSP del processore) diventa quella di P_2 e la **iretq** eseguita all'istante 9 ritorna a P_2 (alla sua prima istruzione, dal momento che P_2 non era ancora mai andato in esecuzione). Successivamente, P_2 invoca anch'esso una primitiva, all'istante 11. All'istante 12 il sistema salva il nuovo stato di P_2 e decide di rimettere in esecuzione P_1 (istante 13). Da dove riparte P_1 ? Dall'ultimo stato salvato, che è sempre quello salvato all'istante 8. Questo stato fa ripartire P_1 dall'istruzione successiva alla **int** che aveva eseguito all'istante 7. In particolare, P_1 riparte in stato utente (istante 14), come se fosse tornato adesso dalla primitiva che aveva

invocato in 7.

All'istante 15 il processore accetta una interruzione esterna e si porta in modo sistema. Anche questa volta viene salvato il nuovo stato di P_1 . In questo caso, il RIP salvato in pila sistema punta all'istruzione successiva all'ultima eseguita prima dell'accettazione dell'interrupt (si ricordi che il processore ascolta gli interrupt solo dopo aver terminato una istruzione). In 17 il sistema termina la gestione dell' interrupt e torna in modo utente senza cambiare processo, quindi la **iretq** torna in P_1 in 18.

Si noti che anche la routine di interruzione usa la pila sistema del processo che si trovava in esecuzione al momento dell'accettazione dell'interruzione.

Realizzazione delle primitive

G. Lettieri

4 Maggio 2021

Vediamo ora come le primitive sono realizzate nel sistema didattico.

1 Atomicità

Le primitive del nostro sistema devono lavorare su un insieme di strutture dati globali, come i descrittori di processo e le code dei processi. Che succederebbe se, mentre una primitiva sta lavorando su una di queste strutture, sia S , una interruzione (sia essa una interruzione esterna, una eccezione o una INT) causasse un salto ad un'altra primitiva, la quale cercasse di accedere alla stessa struttura S ? Pensiamo, per esempio, ad una primitiva che sta cercando di inserire un nuovo `des_proc*`, sia $d1$, in lista `pronti`, supponiamo in testa. Per farlo deve modificare due puntatori: copiare `pronti` in $d1->puntatore$ e scrivere $d1$ in `pronti`. Supponiamo che, tra la prima e la seconda scrittura, il processore salti, per effetto di una interruzione, ad un'altra routine di sistema, e che questa cerchi di inserire un altro `des_proc*`, sia $d2$, in testa alla coda `pronti`. Questa seconda primitiva copierà `pronti` in $d2->puntatore$ e scriverà $d2$ in `pronti`. Al termine della seconda primitiva si ritroverà alla prima, che proseguirà dal punto in cui si era interrotta, e in particolare scriverà $d1$ in `pronti`, cancellando quanto vi aveva scritto la seconda. L'effetto è che il `des_proc* d2` non è più puntato da niente.

Chiaramente non vogliamo che quanto appena descritto possa accadere, ma questo è solo uno degli infiniti problemi che si potrebbero presentare (si pensi, per esempio, ad una `salva_stato` interrotta da un'altra `salva_stato`). Più in generale, quello che abbiamo descritto è un problema di “interferenza” tra due flussi di esecuzione che lavorano su una stessa struttura dati. In generale, noi vogliamo che ogni struttura dati si trovi in uno stato “consistente”. Per esempio, una lista è in uno stato consistente se tutti i suoi elementi sono raggiungibili dalla testa. In un sistema che non prevede interruzioni, le operazioni che manipolano una struttura dati vengono scritte assumendo che la struttura dati si trovi sempre in uno stato consistente quando l'operazione inizia, e assicurandosi di portarla in un nuovo stato consistente *alla fine* dell'operazione. Nel mezzo dell'operazione, però, la struttura dati può passare temporaneamente attraverso stati non consistenti. Ripensiamo all'operazione di inserimento in testa alla coda `pronti` eseguita dalla prima primitiva: subito dopo la co-

pia di `pronti` in `d1->puntatore`, la lista non è in uno stato consistente, in quanto `d1` ne fa concettualmente parte, ma non è ancora puntato da `pronti`. Questo stato inconsistente non è un problema in un sistema senza interruzioni, in quanto non è osservabile da nessun'altra operazione sulla coda: la routine di inserimento proseguirà scrivendo `d1` in `pronti`, riportando così la lista in uno stato consistente. In presenza di interruzioni, però, lo stato inconsistente diventa improvvisamente visibile da un'altra operazione, che era stata scritta assumendo che ciò non potesse mai accadere, e che dunque non è preparata per affrontare la situazione.

Abbiamo due modi principali per evitare i malfunzionamenti causati dall'interferenza:

- scrivere tutte le routine in modo da tener conto di tutti i modi in cui queste si possono mescolare, in modo che funzionino in ogni caso (possibile e anzi desiderabile, ma molto complesso);
- prevenire *a priori* l'interferenza, eliminando tutte le sorgenti di interruzione durante l'esecuzione delle primitive (o almeno delle parti critiche di esse).

Noi adotteremo la seconda soluzione per tutte le primitive del modulo `sistema`, e per la loro intera durata. In particolare, i gate che portano a tali primitive saranno di tipo “interrupt”, con disabilitazione automatica delle interruzioni esterne mascherabili. Durante la scrittura delle primitive, poi, staremo attenti a non causare eccezioni e a non chiaramare altre primitive tramite `int`. Con questi accorgimenti, le nostre primitive gireranno in un contesto “atomico”: una volta iniziate saranno portate a compimento, senza che niente le possa interrompere¹. Diventeranno in questo modo molto simili alle singole istruzioni di linguaggio macchina, la cui atomicità è garantita dal processore (che gestisce interruzioni esterne e eccezioni solo tra una istruzione e la successiva).

Si noti che in molti sistemi reali l'atomicità viene considerata un prezzo troppo alto da pagare e viene rilassata in vari modi. Noi la rilasseremo solo nel modulo `io`, ma alcuni testi d'esame considerano il caso di rilassamento anche nel modulo `sistema`.

2 Meccanismo di chiamata

A livello Assembler, invocare una primitiva non è come invocare una semplice funzione in quanto, come abbiamo detto, è necessario passare attraverso un gate della IDT con una istruzione `int`.

Al livello del C++, però, possiamo fare in modo che la primitiva si usi come una qualunque funzione, per maggior comodità dell'utente. Il meccanismo che usiamo è illustrato in Figura 1. L'utente, nel file `utente.cpp`, dichiara e chiama la funzione `primitiva_i()`, con l'obiettivo di eseguire la funzione

¹Salvo bug e interruzioni esterne non mascherabili, che comunque causeranno il blocco dell'intero sistema.

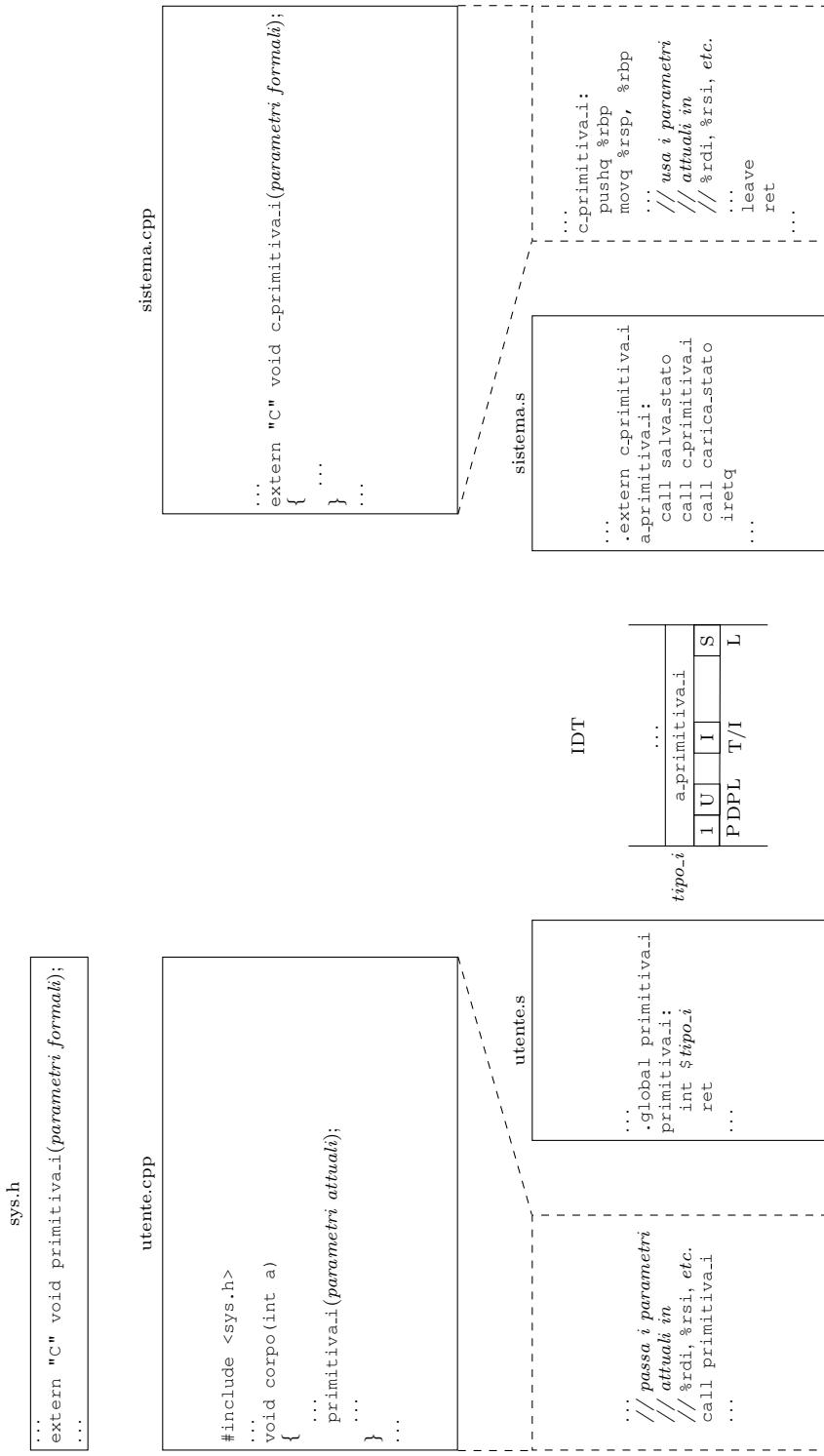


Figura 1: Primitive che possono causare un cambio di processo.

`c_primitiva_i()` che è contenuta nel modulo sistema. La `primitiva_i()` è in realtà solo un piccolo programma di interfaccia, scritto in assembler e contenuto nel file `utente.s`, che si limita ad eseguire l'istruzione `int` con l'indice del gate della primitiva nella IDT (*tipo_i*).

L'istruzione `int` si preoccuperà di innalzare il livello di privilegio (con le varie operazioni connesse, tra cui il cambio di pila) e saltare al codice della primitiva nel modulo sistema, salvando il pila l'indirizzo dell'istruzione successiva (una `ret`, in questo caso). Se non ci sarà un cambio di processo, questa è l'istruzione a cui il processore salterà al termine dell'esecuzione della primitiva.

Si noti che, a livello Assembler, anche *ritornare* da una primitiva non è come ritornare da una normale funzione, in quanto è necessario eseguire l'istruzione `iretq`, che è l'unica che permette di riportare il processore a livello utente. Anche nel modulo sistema dovremo quindi aggiungere una funzione di interfaccia (`a_primitiva_i` in Figura), che chiami la `c_primitiva_i()` e poi esegua la `iretq`. Con questo accorgimento, la `c_primitiva_i()` può essere scritta in C++ e compilata normalmente.

Affinchè l'istruzione “`int $tipo_i`” salti alla funzione `a_primitiva_it`, con innalzamento di privilegio, il programmatore di sistema deve predisporre l'entrata della IDT di offset *tipo_i* come illustrato in figura:

- il campo IND (indirizzo a cui saltare) contiene `a_primitiva_i`;
- il campo P (gate Present) contiene 1, ad indicare che il gate è implementato;
- il campo DPL (Descriptor Privilege Level) indica che il gate può essere utilizzato da livello utente tramite una istruzione `int`;
- il campo L (Level) indica che, dopo il salto, il processore si deve trovare a livello sistema;
- per i motivi che abbiamo visto nella sezione precedente, il campo I/T indica che il gate è di tipo “interrupt”, in modo che le interruzioni esterne mascherabili siano disabilitate.

La `a_primitiva_i` dovrà anche chiamare `salva_stato` e `carica_stato`, per realizzare il meccanismo del cambio di processo. In questo modo la funzione `c_primitiva_i()` può sospendere il processo corrente e schedularne un altro, semplicemente cambiando il valore della variabile `esecuzione`.

I parametri formali della `c_primitiva_i()` sono gli stessi della corrispondente `primitiva_i()`. In Figura 1 si vede che, nel tradurre la chiamata `a_primitiva_i()`, il compilatore C++ copierà i parametri attuali nei registri `rdi`, `rsi`, etc. Questi registri non vengono modificati mentre si passa da `primitiva_i` a `a_primitiva_i`, quindi la funzione `c_primitiva_i()` li troverà ancora lì, dove il compilatore C++ se li aspetta.

Sia `primitiva_i()` che `c_primitiva_i()` sono dichiarate `extern "C"`. In questo modo il compilatore C++ assume che le due funzioni seguano lo standard di aggancio del linguaggio C, che non prevede l'overloading delle funzioni e

dunque non richiede che i nomi delle funzioni vengano trasformati come abbiamo visto nel caso del C++. Facciamo questo per comodità, dal momento che non prevediamo di sfruttare l'overloading per le primitive.

Normalmente il programmatore di sistema fornisce all'utente anche il file `utente.s` e un header file che contenga le dichiarazioni delle primitive (`primitiva_i()`, nell'esempio). L'utente non deve fare altro che includere tale file, con la direttiva `#include`, nel suo `utente.cpp`. Nel nostro caso le dichiarazioni si trovano nel file `sys.h` (nella directory `utente/include`).

2.1 Primitive che restituiscono un risultato

Si noti che la primitiva in Figura 1 è di tipo `void`. La presenza della chiamata di `carica_stato` rende un po' complicato restituire un valore dalla `c_primitiva_i()` alla `primitiva_i()` tramite il registro `rax`. Una istruzione di “`return ris;`” nella `c_primitiva_i()` verrebbe tradotta dal compilatore C++ lasciando il valore `ris` nel registro `rax`, ma la `carica_stato` sovrascriverebbe tale valore prima che la `primitiva_i()` possa vederlo.

Se una primitiva deve restituire un valore, occorre operare come in Figura 2. La funzione `primitiva_j()`, che è quella direttamente invocata dall'utente, è dichiarata di tipo `tipo_r`, ma la corrispondente `c_primitiva_j()` è `void`. Il valore `ris` deve essere restituito modificando il campo `contesto[I_RAX]` del descrittore del processo, in modo che la successiva `carica_stato` ricopi `ris` nel registro `rax`, dove se lo aspetta il compilatore C++ dopo la chiamata a `primitiva_j()`.

2.2 Primitive che non causano cambi di processo

Se una primitiva non causa mai un cambio di processo, alcune cose possono essere semplificate come in Figura 3. Il programmatore di sistema può eliminare le chiamate a `salva_stato` e `carica_stato` in `c_primitiva_k`. Inoltre, se la primitiva deve restituire un valore, può tranquillamente lasciarlo in `rax`, in quanto ora non verrà sovrascritto. In Figura 3 vediamo che `c_primitiva_k()` è ora dichiarata di tipo `tipo_r`, come la corrispondente `primitiva_k()` e il risultato `ris` può essere restituito con una normale `return`.

Si noti però che, con questo schema, eventuali registri *scratch* sporcati dalla primitiva non verrebbero ripristinati al ritorno a livello utente. Questa può essere considerata una violazione della riservatezza dei dati del sistema.

3 Scrivere nuove primitive

Per aggiungere una nuova primitiva si deve seguire lo schema appropriato di Figura 1, 2 o 3 e predisporre una entrata della IDT.

Supponiamo di voler aggiungere una primitiva `getid()`, senza parametri, che restituisce l'identificatore del processo che la invoca.

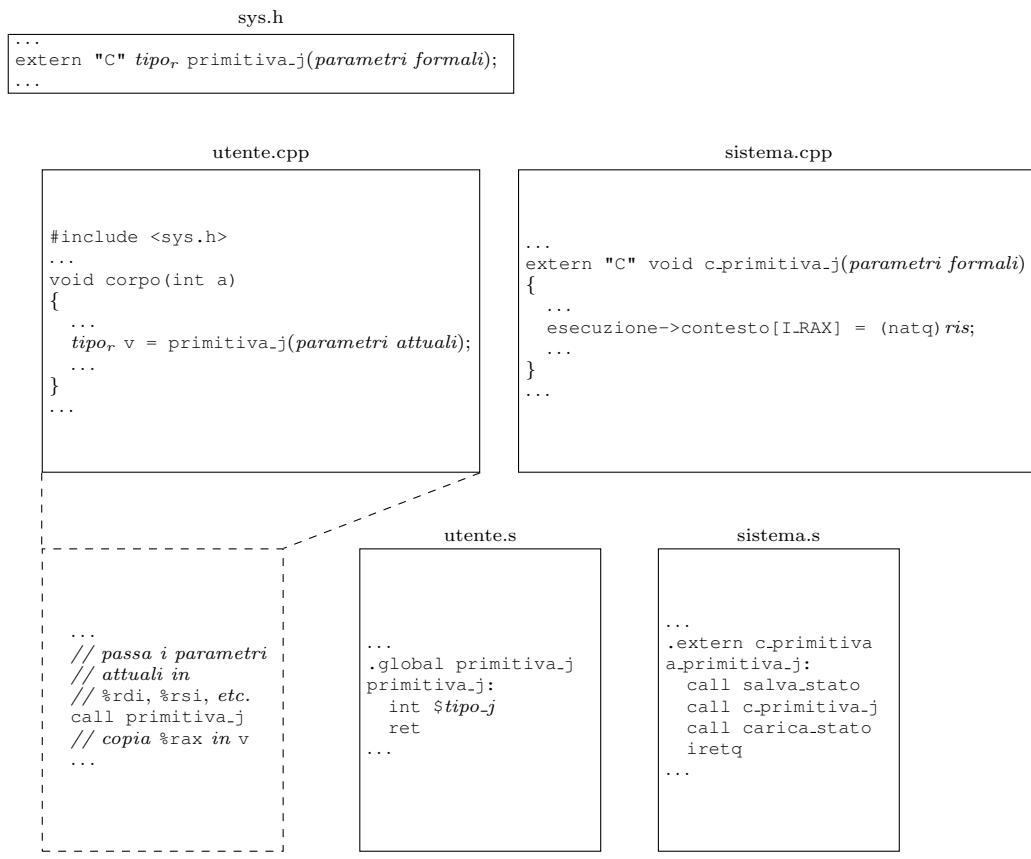


Figura 2: Primitive che possono causare un cambio di processo e devono restituire un valore.

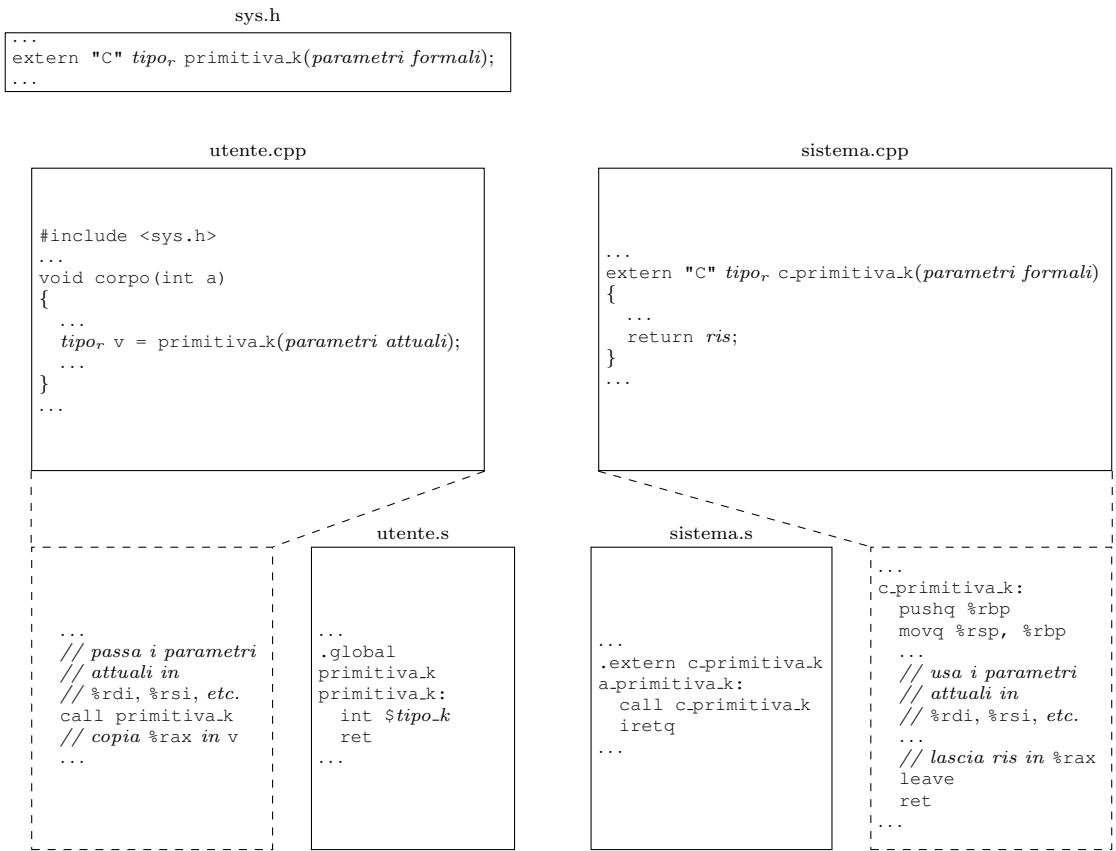


Figura 3: Primitive che non causano mai un cambio di processo.

Per prima cosa occorre assegnare un tipo di interruzione alla primitiva. I tipi di tutte le primitive già realizzate sono definiti nel file `costanti.h` nella cartella `include`, con nomi che iniziano con `TIPO_`. Si noti che i tipi sono definiti come macro, in modo che il file possa essere utilizzato sia dal C++ che dall’assembler. Sceglieremo un tipo non utilizzato (per esempio, `0x2a`) e definiamo una nuova macro:

```
#define TIPO_GETID 0x2a
```

Per caricare il corrispondente gate della IDT possiamo aggiungere una riga alla funzione `init_idt` che si trova nel file `sistema.s`. Tale funzione è chiamata all’avvio del sistema e si occupa di inizializzare la tabella IDT. Possiamo usare la macro `carica_gate` che richiede tre parametri:

- il tipo della primitiva (da cui si deduce il gate della IDT da inizializzare);
- l’indirizzo a cui saltare quando qualcuno usa il gate;
- il livello di privilegio minimo richiesto per utilizzare il gate tramite una `int` (campo DPL del gate).

La macro inizializza sempre il campo P con 1 (gate implementato), il campo I/T con I (gate di tipo interrupt) e il campo L con S (la primitiva andrà in esecuzione a livello sistema). Nel nostro caso aggiungeremo la riga

```
carica_gate TIPO_GETID a_getid LIV_UTENTE
```

dove `LIV_UTENTE` è una costante che specifica il livello utente (esiste anche la costante `LIV_SISTEMA` per indicare che il gate può essere usato solo da livello sistema).

Sempre nel file `sistema.s` dobbiamo scrivere la funzione `a_getid`. In questo caso possiamo adottare lo schema di Figura 3 (questa primitiva non ha sicuramente bisogno di bloccare il processo che la chiama):

```
.extern c_getid
a_getid:
    call c_getid
    iretq
```

Infine, scriviamo la primitiva vera e propria (in `sistema.cpp`):

```
extern "C" natl c_getid()
{
    return esecuzione->id;
}
```

Dal punto di vista del modulo sistema non dobbiamo fare altro. Possiamo ricompilare il modulo sistema con il comando “make” e correggere eventuali errori di sintassi.

Il programmatore di sistema, come detto, dovrebbe anche fornire la dichiarazione e il programma assembler di interfaccia per l'utente. In sys.h aggiungiamo

```
extern "C" natl getid();
```

e nel file utente.s

```
.global getid
getid:
    int $TIPO_GETID
    ret
```

Il compito del programmatore di sistema finisce qui. A questo punto gli utenti possono scrivere i loro programmi e usare la nuova primitiva. Per esempio, scriviamo il seguente programma utente nel file utente.cpp nella directory utente:

```
#include <all.h>

void corpo(int a)
{
    natl id;

    id = getid();
    printf("Il mio id: %d", id);
    terminate_p();
}

int main()
{
    activate_p(corpo, 0, 20, LIV_UTENTE);
    terminate_p();
}
```

Per provare il programma utente lanciamo il comando “make” (correggendo eventuali errori di sintassi) e poi avviamo il sistema con “./run”.

3.1 Funzioni di supporto

Le seguenti funzioni sono già definite in sistema.cpp e possono essere utilizzate nel definire nuove primitive.

des_proc *des_p(natl id)

Restituisce un puntatore al descrittore del processo di identificatore id (nullptr se tale processo non esiste).

void schedulatore()

Sceglie il prossimo processo da mettere in esecuzione (cambia il valore della variabile esecuzione).

```

void inserimento_lista(des_proc *&p_list, des_proc *p_elem)
    Inserisce p_elem nella lista p_list, mantenendo l'ordinamento basato
    sul campo precedenza. Se la lista contiene altri elementi che hanno la
    stessa precedenza del nuovo, il nuovo viene inserito come ultimo tra questi.

des_proc* rimozione_lista(des_proc *&p_list)
    Estrae l'elemento in testa alla p_list e ne restituisce un puntatore
    (nullptr se la lista è vuota).

void inspronti()
    Inserisce il des_proc puntato da esecuzione in testa alla coda pronti.

void c_abort_p()
    Distrugge il processo puntato da esecuzione e chiama schedulatore().

```

3.2 Considerazioni generali

Discussiamo qui alcune cose a cui prestare attenzione nella scrittura delle primitive.

- Come abbiamo visto, le primitive girano in contesto atomico. Quindi, durante l'esecuzione una primitiva, il processore non fa altro che eseguire il codice della primitiva stessa: niente altro avviene concorrentemente nella CPU. In particolare, se eseguiamo un ciclo infinito in una primitiva, tutto il sistema si ferma. È un grave errore scrivere una cosa del genere nel codice di una primitiva:

```

int i = 0;
while (i == 0) {
    // altre cose che non modificano i
}

```

Nessuno può modificare il valore della variabile `i`, quindi questo è un ciclo infinito.

- Cambiare il valore della variabile `esecuzione` non cambia magicamente il processo corrente. Il salto al nuovo processo avverrà solo quando verrà chiamata la funzione `carica_stato`, seguita dall'istruzione `iretq`.
- La (nostra) funzione `schedulatore()` non fa altro che cambiare il valore della variabile `esecuzione` (quindi non salta lei stessa ad un nuovo processo).
- Se un processo viene sospeso mentre esegue una primitiva (mettendo qualche altro processo in `esecuzione` al suo posto), al suo suo risveglio riparte dall'istruzione successiva alla `int` con cui aveva chiamato la primitiva. In generale riparte dall'ultimo stato salvato e, nel caso di sospensione durante l'esecuzione di una primitiva, l'ultimo stato è quello salvato dalla `salva_stato` all'inizio della parte assembler della primitiva.

Semafori

G. Lettieri

4 Maggio 2021

Nel sistema didattico tutti i processi utente condividono le sezioni **.text**, **.data**, **.bss** del modulo utente, e lo heap. Ciascun processo ha invece una sua pila utente privata, inaccessibile agli altri processi. Questo tipo di condivisione può essere ottenuta evitando di rimpiazzare la parte di memoria condivisa quando si salta da un processo ad un altro.

Un sistema del genere ha senso se i processi appartengono tutti allo stesso utente e fanno parte di un'unica applicazione, che l'utente ha deciso di strutturare in più attività concorrenti. Da ora in poi ci limiteremo a considerare solo questo caso.

L'utente che scrive una applicazione strutturata su più processi concorrenti deve affrontare dei problemi, che in parte abbiamo già visto, perché sono molto simili a quelli già affrontati a livello sistema.

In particolare, anche l'utente deve afrontare il problema dell'*interferenza*. Mentre un processo sta eseguendo delle modifiche su una struttura dati comune, un altro processo potrebbe inserirsi e cominciare anche lui a modificare la *stessa* struttura dati. Se l'utente non scrive il codice con attenzione, questo può causare malfunzionamenti, come abbiamo visto.

Si noti che nel codice di sistema abbiamo risolto il problema ricorrendo all'*atomicità*, realizzata disabilitando le interruzioni mentre è in esecuzione codice di sistema. Qui non possiamo fare la stessa cosa, in quanto le interruzioni devono restare abilitate mentre è in esecuzione il codice utente (altrimenti non riusciamo a realizzare un sistema multiprogrammato), e non possiamo dare all'utente la possibilità di disabilitare e riabilitare le interruzioni a suo piacimento, perché potrebbe disabilitarle e non riabilitarle più.

L'*atomicità* è un caso estremo di *mutua esclusione*. Con questa locuzione ci si riferisce alla proprietà secondo la quale alcune operazioni non possono mescolarsi tra loro (si escludono a vicenda). Quello che vogliamo fare è fornire al nostro utente delle primitive tramite le quali possa realizzare la mutua esclusione delle procedure che accedono alle sue strutture dati, senza però compromettere il funzionamento di tutto il sistema.

Esaminiamo ora il problema, diverso, della *sincronizzazione* tra i processi. Nello strutturare la sua applicazione, l'utente può aver bisogno di fare in modo che una certa azione *B* avvenga sempre dopo una certa azione *A*. Se però le due azioni sono svolte da processi diversi, l'utente ha il problema di non poter prevedere quando questi due processi svolgeranno le loro azioni. Un caso comune

è quello in cui un processo produce dei dati e li scrive in un buffer intermedio, da cui un altro processo li preleva per svolgere ulteriori elaborazioni. Normalmente questi due processi sono ciclici, con il produttore che produce continuamente nuovi dati e il consumatore che li preleva continuamente. Il buffer è una zona di memoria che può contenere un numero limitato di dati—supponiamo uno solo, per semplicità. La scrittura di un nuovo dato sovrascrive dunque il vecchio. L'utente vorrebbe poter garantire che il produttore non possa sovrascrivere un dato che il consumatore non ha ancora letto, e che il consumatore non legga mai due volte lo stesso dato.

Si noti come i problemi di sincronizzazione siano diversi da quelli di mutua esclusione: nella sincronizzazione vogliamo garantire un ordinamento tra alcune azioni, che deve essere sempre rispettato. Non è così nella mutua esclusione. Si pensi a due processi, siano P_1 e P_2 , che vogliono inserire un elemento nella stessa lista. L'unica cosa che non vogliamo è che i due inserimenti vengano tentati contemporaneamente. Se, per esempio, P_1 ha iniziato un inserimento, P_2 deve aspettare che P_1 abbia finito prima di iniziare il suo. Questa *sembra* la stessa cosa della sincronizzazione, ma dobbiamo tenere presente che ci va bene anche il contrario: se P_2 arriva prima e comincia il suo inserimento, è P_1 che deve aspettare che P_2 abbia finito. Inoltre, nessuno dei due deve aspettare niente, se quando vuole fare l'inserimento l'altro non sta usando la lista. Non stiamo dunque stabilendo un ordinamento tra le operazioni di P_1 e P_2 .

1 Scatole e gettoni

Per risolvere i problemi di mutua esclusione e sincronizzazione, supponiamo di avere delle scatole che possono contenere degli oggetti, tutti uguali, che chiamiamo gettoni. Su queste scatole possono essere eseguite solo due operazioni:

- inserire un gettone—non è necessario che il gettone sia stato precedentemente preso da una scatola;
- prendere un gettone—se non ce ne sono, bisogna aspettare che qualcuno ne inserisca uno, *senza poter fare nient'altro*.

Supponiamo di dover risolvere un problema di mutua esclusione: abbiamo più persone, P_1, P_2, \dots, P_n , e un corrispondente numero di azioni A_1, A_2, \dots, A_n che queste persone devono compiere. Vogliamo che le azioni non possano mai essere eseguite contemporaneamente. È sufficiente avere una scatola che inizialmente contiene un solo gettone e imporre la regola che solo chi ha il gettone può compiere una delle azioni: chiunque voglia agire, dunque, deve prima prendere un gettone e poi rimetterlo nella scatola quando ha finito. Dal momento che c'è un solo gettone, non è possibile che ci siano due azioni contemporaneamente in corso. Se qualcuno vuole iniziare una azione mentre ne è in corso un'altra, troverà la scatola vuota e dovrà aspettare che l'altro abbia finito.

Si noti che, mentre è in corso una azione, supponiamo da parte di P_1 , tante persone potrebbero volerne iniziare un'altra. Tutte queste dovranno aspettare.

Quando P_1 avrà finito poserà il suo gettone, che verrà preso da una delle persone in attesa, sia P_2 . Quando P_2 avrà a sua volta finito, il gettone passerà ad un'altra persona ancora, e così via. Si noti anche che la soluzione richiede una completa collaborazione da parte di tutti i partecipanti: se qualcuno compie una azione senza prendere il gettone, o si dimentica di rimetterlo a posto dopo aver finito, il sistema non funziona.

Vediamo ora come risolvere un problema di sincronizzazione. Abbiamo due persone, P_a che deve compiere l'azione A e P_b che deve compiere l'azione B . Vogliamo che l'azione B sia eseguita sempre dopo l'azione A . È sufficiente prevedere una scatola inizialmente vuota. Vogliamo che la presenza di un gettone nella scatola rappresenti il fatto che A è stata eseguita e B ancora no. Operativamente, *dopo* aver eseguito l'azione A , P_a deve lasciare un gettone nella scatola; *prima* di eseguire l'azione B , P_b deve prendere un gettone dalla scatola. Se P_a arriva per primo alla scatola, lascia il gettone e poi P_b potrà prenderlo e proseguire. Se invece arriva per primo P_b , trova la scatola vuota e deve aspettare che P_a inserisca un gettone. In entrambi i casi l'azione B non potrà partire prima che sia finita l'azione A .

In generale, per utilizzare le scatole di gettoni per risolvere un problema particolare, bisogna specificare quante scatole servono e quanti gettoni contengono inizialmente, quindi stabilire delle regole su chi deve inserire/estrarre gettoni, e quando. Il problema si può considerare risolto solo se le regole sono scelte bene e tutti le rispettano.

1.1 Esercizio

Come risolvere il problema del produttore e consumatore? Il produttore P invia dei messaggi al consumatore C tramite una lavagna. Per scrivere un nuovo messaggio cancella il precedente. Inoltre, non c'è modo di distinguere un messaggio nuovo da uno vecchio semplicemente guardandoli. Come fare per garantire che C legga tutti i messaggi di P e non legga mai due volte lo stesso messaggio?

2 Semafori

Nel nostro sistema, forniremo agli utenti l'astrazione delle scatole di gettoni, chiamate però, per motivi storici, *semafori*. Ogni semaforo è identificato da un numero. Forniamo le seguenti primitive di sistema:

- **nat1 sem_ini(nat1 v):** crea un nuovo semaforo, che inizialmente contiene v gettoni, e ne restituisce l'identificatore (0xFFFFFFFF se non è stato possibile crearlo);
- **void sem_wait(nat1 sem):** prende un gettone dal semaforo numero sem ; blocca il processo se il semaforo è vuoto;
- **void sem_signal(nat1 sem):** inserisce un gettone nel semaforo sem ; risveglia uno dei processi bloccati in attesa di un gettone, se ve ne sono.

Si noti che, nonostante il nome, la primitiva `sem_wait()` non causa necessariamente una attesa con blocco del processo in esecuzione: se la scatola contiene già un gettone, il processo lo prende e va avanti senza bloccarsi.

Per semplicità, non diamo la possibilità di distruggere semafori. Dal momento che abbiamo deciso di supportare solo il caso di un singolo utente che esegue una applicazione multi-processo, assumiamo che tutti i semafori creati servano fino al termine dell'applicazione.

2.1 Mutua esclusione

Il problema della mutua esclusione tra le azioni A_1, A_2, \dots, A_n dei processi P_1, P_2, \dots, P_n può essere risolto come abbiamo visto nella Sezione 1. Prima che i processi siano attivati, l'utente deve creare un semaforo inizializzato ad 1:

```
nat1 mutex = sem_ini(1);
```

Quindi, tutte le azioni A_i , $1 \leq i \leq n$, devono essere protette dal semaforo:

$P_i:$	<code>sem_wait(mutex);</code>
	$A_i;$
	<code>sem_signal(mutex);</code>

Se, per esempio, le azioni A_i sono tutte invocazioni di funzioni membro di una classe su uno stesso oggetto, è sufficiente aggiungere l'identificatore del semaforo ai membri della classe, creare il semaforo nel costruttore, quindi aggiungere le chiamate `sem_wait()` e `sem_signal()` ad ogni funzione membro. In questo modo tutte le operazioni sullo stesso oggetto verranno eseguite in mutua esclusione. Se chiamiamo “libero” il semaforo, possiamo leggere l'operazione `sem_wait(libero)` come “attendi the l'oggetto sia libero (nessun altro lo sta usando)” e `sem_signal(libero)` come “segnala che ora l'oggetto è libero”.

2.2 Sincronizzazione

Il problema della sincronizzazione tra due processi— P_a che deve compiere l'azione A e P_b che deve compiere l'azione B —può essere risolto creando, prima che i processi siano attivati, un semaforo inizializzato a zero:

```
nat1 sync = sem_ini(0);
```

Quindi, i due processi devono essere codificati nel seguente modo:

$P_a:$	<code>A;</code>
	<code>sem_signal(sync);</code>
$P_b:$	<code>sem_wait(sync);</code>
	$B;$

Se P_b sta aspettando il verificarsi di una condizione, inizialmente falsa e resa vera dall'azione A , è utile dare al semaforo un nome x che ricordi questa condizione

(per esempio, `nuovo_messaggio`). Allora l'azione di P_b si legge come “aspetto che la condizione x sia vera” e l'azione di P_a si legge come “segnalo che la condizione x ora è vera”. Anche in questo caso non è detto che la `sem_wait()` debba sempre aspettare (bloccare il processo): se P_a completa l'azione A ed esegue la `sem_signal()` prima che P_b arrivi alla sua `sem_wait()`, P_b troverà il gettone già nella scatola e potrà prenderlo senza dover aspettare niente.

2.3 Realizzazione dei semafori

Per realizzare i semafori prevediamo la seguente struttura dati, definita nel codice di sistema:

```
struct des_sem {
    int counter;
    des_proc *pointer;
};
```

Il campo `counter` conta i gettoni contenuti nel semaforo, se maggiore o uguale a zero. Permettiamo al campo `counter` di scendere anche sotto zero. In questo modo la `sem_wait()` può decrementarlo in ogni caso. Se è negativo, il suo valore assoluto indica quanti processi sono in attesa di un gettone. I processi in attesa sono inseriti nella lista `pointer`. Queste liste realizzano le code dei processi bloccati in attesa di un “evento”. L’evento non è altro che l’inserimento di un gettone nel semaforo. I processi bloccati su un semaforo verranno rimessi in coda pronti quando riceveranno un gettone, per effetto di un altro processo che invoca `sem_signal()` sullo stesso semaforo. La primitiva `sem_signal()` deve incrementare il numero di gettoni e, se ci sono processi nella lista `pointer`, estrarne uno e inserirlo in coda pronti. Si noti che abbiamo deciso di realizzare un sistema in cui ogni processo ha una priorità, e ora dobbiamo garantire che in esecuzione ci sia sempre il processo che ha maggiore priorità tra tutti quelli pronti. Se la `sem_signal()` inserisce un processo in coda pronti, deve anche controllare che questo processo non abbia priorità maggiore di quello attualmente in esecuzione (quello che ha invocato la `sem_signal()`). In tal caso, il processo corrente deve andare in coda pronti (preemption) e quello appena risvegliato deve andare direttamente in esecuzione.

Come tutte le primitive, anche `sem_ini()`, `sem_wait()` e `sem_signal()` sono invocate tramite una istruzione `int` che, se eseguita da livello utente, causa un innalzamento del livello di privilegio del processore e un salto alla parte assembler della primitiva, che salva lo stato del processo corrente e chiama la parte C++, poi carica lo stato del processo puntato da `esecuzione` (che potrebbe essere cambiato) ed esegue una `iretq`.

La Figura 1 mostra la parte C++ della primitiva `sem_ini`. Per l’allocazione dei semafori riserviamo un array di strutture `des_sem` e ci limitiamo ad usarne una nuova, presa dall’array, ogni volta che l’utente ci chiede un nuovo semaforo. L’array ci permetterà di risalire facilmente alla struttura `des_sem` corretta, quando l’utente passerà l’identificatore del semaforo alle primitive `sem_wait()` e `sem_signal()`. Come vedremo, anche il modulo `io` utilizza dei semafori.

```

1 des_sem array_dess[MAX_SEM * 2];
2 extern "C" void c_sem_ini(int v)
3 {
4     natl i = alloca_sem();
5     if (i != 0xFFFFFFFF)
6         array_dess[i].counter = val;
7     esecuzione->contesto[I_RAX] = i;
8 }
```

Figura 1: Parte C++ della primitiva `sem_ini()`.

```

1 extern "C" void c_sem_wait(natl sem)
2 {
3     des_sem *s = &array_dessem[sem];
4     s->counter--;
5     if (s->counter < 0) {
6         inserimento_lista(s->pointer, esecuzione);
7         schedulatore();
8     }
9 }
```

Figura 2: La parte C++ della primitiva `sem_wait()`.

Per evitare che l’utente possa erroneamente usare i semafori allocati dal modulo `io`, i semafori vengono idealmente distinti in “utente” e “sistema”. I semafori utente sono quelli che si trovano nelle prime `MAX_SEM` posizioni dell’array, e i rimanenti sono di livello sistema. La funzione `alloca_sem()` allocherà un indice appartenente ad una delle due parti dell’array, in base al livello di privilegio che il processo aveva quando ha invocato la primitiva (come salvato dal processore nella pila sistema). La parte C++ della primitiva `sem_wait()` è mostrata in Figura 2. Nel caso di semaforo senza gettoni (linea 5), il processo attualmente in esecuzione viene inserito nella coda del semaforo (linea 6) e ne viene scelto un altro invocando la funzione `schedulatore()`. Questa estrae dalla coda pronti il processo a più alta priorità e lo fa puntare dalla variabile `esecuzione`, in modo che la routine `carica_stato` (che verrà eseguita subito dopo) faccia saltare al nuovo processo, di fatto bloccando il precedente.

La Figura 3, infine, mostra la parte C++ della primitiva `sem_signal()`. Se ci sono processi in coda sul semaforo (linea 5), la primitiva ne estrae uno (linea 6). Si noti che la funzione `rimozione_lista`, nel caso vi fosse più di un processo, estrae quello a maggiore priorità. A questo punto la primitiva deve scegliere chi deve proseguire, tra il processo in esecuzione e quello appena estratto. La cosa più semplice è di inserire entrambi i processi in coda pronti e lasciar scegliere alla funzione `schedulatore()` (linee 7-9).

Sia la `sem_wait()` che la `sem_signal()`, prima di usare `sem`, devono

```

1 extern "C" void c_sem_signal(natl sem)
2 {
3     des_sem *s = &array_dessem[sem];
4     s->counter++;
5     if (s->counter <= 0) {
6         des_proc* lavoro = rimozione_lista(s->pointer);
7         inspronti();
8         inserimento_lista(pronti, lavoro);
9         schedulatore();
10    }
11 }

```

Figura 3: La parte C++ della primitiva `sem_signal()`.

controllare che questo sia un valido identificatore di semaforo, cioè un numero precedentemente restituito da una invocazione di `sem_ini()` per il livello (utente o sistema) corretto, e terminare forzatamente il processo in caso contrario. Questi controlli sono stati omessi dalle Figure 2 e 3 per semplicità, ma sono presenti nel codice del nucleo.

2.4 Utilizzo del debugger

Le estensioni del debugger contengono anche alcuni comandi relativi ai semafori:

`sem` mostra lo stato di tutti i semafori allocati.

`sem waiting`
mostra lo stato di tutti i semafori la cui coda non è vuota.

Il comando `sem waiting` viene eseguito automaticamente ogni volta che il debugger riacquisisce il controllo. Lo stato dei semafori è mostrato nella forma

$\{counter, lista\ processi\}$.

Paginazione

G. Lettieri

12 Maggio 2021

Per quanto visto finora, ogni volta che si esegue un cambio di processo tutto lo stato privato del processo uscente, contenuto in M2, deve essere ricopiato nello swap e sostituito con lo stato del processo entrante, letto dallo swap. In sistemi in cui i processi non hanno parti di memoria condivise, questo può significare che è necessario trasferire da e verso lo swap l'intero contenuto di M2.

Quello che vogliamo fare ora è di eliminare, o quantomeno ridurre, queste copie da e verso lo swap. L'idea di partenza è che la maggior parte dei processi non avrà bisogno di tutta la memoria M2, quindi potremmo pensare di caricare più di uno stato alla volta e di tenere in memoria anche lo stato dei processi che non sono in esecuzione, in modo da averli già pronti quando verranno schedulati. In pratica la memoria M2 si comporterà come una cache dello swap, con gli stessi problemi da risolvere: quando tutta M2 è piena e dobbiamo mettere in esecuzione un processo il cui stato non è in M2, dobbiamo fare spazio togliendo lo stato di qualche altro processo; quale scegliamo? Non ci addentreremo però in questi argomenti, che appartengono al corso di Sistemi Operativi. Ci limitiamo invece a studiare il meccanismo che permette di mantenere in memoria lo stato di più di un processo.

1 Problemi

Affronteremo i problemi gradualmente, in modo da capire le motivazioni dietro il meccanismo che adotteremo alla fine.

1.1 Dimensione massima di ogni processo

Per prima cosa dobbiamo assumere di sapere di quanta memoria ha bisogno ogni processo. Supporremo che sia il programmatore stesso a dirlo al sistema (magari aiutato da compilatore e collegatore). Un processo avrà bisogno di un po' di memoria per contenere la sezione **.text**, contenente il codice del programma da eseguire, e di altra memoria per la sezione **.data**, contenente (in C++) le variabili globali. La dimensione di queste sezioni è nota al collegatore. Ci sono però altre due sezioni, inizialmente vuote, che si possono espandere durante l'esecuzione: la pila e lo heap (usato per la memoria dinamica). Per queste il programmatore deve stabilire un massimo. La memoria di un processo viene

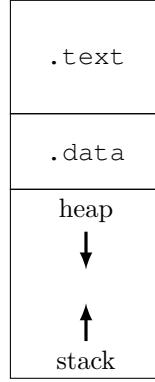


Figura 1: Organizzazione della memoria di un processo. Lo heap e lo stack si trovano ai capi opposti di un'unica regione di memoria, con lo heap che si espande verso il basso e lo stack verso l'altro.

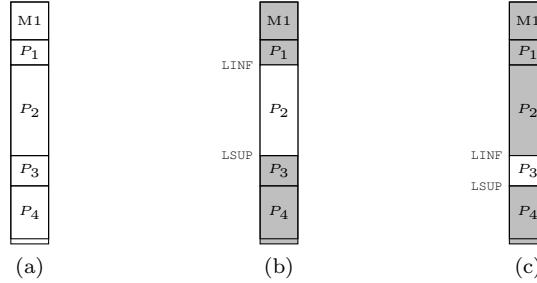


Figura 2: Un esempio di evoluzione del sistema. (a) vengono caricati P_1 , P_2 , P_3 e P_4 ; (b) quando è in esecuzione P_2 non si può accedere alla memoria degli altri processi; (c) cambiando il contenuto di LINF e LSUP cambia la zona di memoria accessibile.

tipicamente organizzata come in Figura 1, con tutte le sezioni contigue e con lo heap e la pila che condividono una stessa zona di memoria. È sufficiente che il programmatore dica al sistema quanto deve essere grande la zona utilizzata dallo heap e dalla pila. Il compilatore, il collegatore, o anche il sistema stesso, possono assumere un valore di default, per non costringere il programmatore a specificare questa dimensione nei casi più comuni.

Sapendo quanto è grande ogni processo, il sistema può caricarne in memoria più di uno, come in Figura 2(a).

1.2 Isolamento tra i processi

Il secondo problema che dobbiamo risolvere è: come impedire che lo stato dei processi non in esecuzione venga letto o modificato dal processo in esecuzione?

Per il momento abbiamo previsto nella CPU un meccanismo che protegge la memoria M1, ma non la memoria M2. Ricordiamo in cosa consiste questo meccanismo:

- abbiamo introdotto un registro, scrivibile solo da livello sistema, che contiene un indirizzo *limite*;
- abbiamo modificato la CPU in modo che, quando si trova a livello utente, controlli che ogni operazione in memoria (prelievo istruzioni e lettura/-scrittura operandi) avvenga a indirizzi maggiori del limite, sollevando una eccezione in caso contrario.
- il registro limite viene inizializzato con l'ultimo indirizzo di M1 all'avvio del sistema.

Si vede che questo meccanismo è efficace nel proteggere M1, ma non impedisce in alcun modo al processo in esecuzione di accedere ovunque voglia nella memoria M2, dunque anche nelle parti che contengono lo stato degli altri processi.

Possiamo pensare di aggiungere altri due registri alla CPU, chiamiamoli LINF (limite inferiore) e LSUP (limite superiore). Nel descrittore di ogni processo prevediamo spazio anche per questi due nuovi registri, `contesto[I_LINF]` e `contesto[I_LSUP]`.

- I due nuovi registri sono scrivibili solo da livello sistema.
- Quando si trova a livello utente, la CPU controlla che ogni accesso in memoria abbia un indirizzo compreso tra LINF e LSUP (e solleva una eccezione in caso contrario).
- Ogni volta che un processo viene caricato dallo swap (la prima volta, o dopo che era stato rimosso per fare spazio) il sistema deve inizializzare i campi `contesto[I_LINF]` e `contesto[I_LSUP]` con l'indirizzo iniziale e finale della parte di M2 occupata dal processo.
- Ogni volta che si cambia processo, si aggiorna anche il contenuto di LINF e LSUP con i valori presi dal descrittore del processo entrante.

Questi due registri risolvono effettivamente il problema: ogni processo non può accedere al di fuori della propria zona di memoria (Figura 2(b) e (c)). Questi due registri rendono anche inutile il registro limite che avevamo introdotto prima: a maggior ragione, i processi utente non possono accedere alla memoria M1.

1.3 Caricamento a indirizzi variabili

Il sistema così realizzato presenta però degli svantaggi. Prima tutti i processi venivano sempre caricati a partire dallo stesso indirizzo (l'inizio di partenza di M2), mentre ora possono essere caricati ovunque, in base a quali altri processi si trovano già in memoria al momento del caricamento. L'indirizzo di caricamento non è dunque noto durante la compilazione e il collegamento: come scegliere

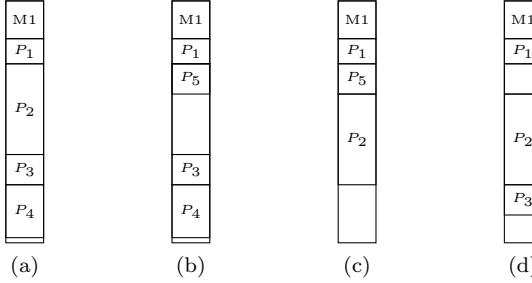


Figura 3: (a) ultimo stato dalla Figura 2; (b) P_2 viene temporaneamente rimosso per far posto a P_5 ; (c) P_3 e P_4 vengono temporaneamente rimosso per far posto a P_2 ; (d) viene ricaricato P_3 e P_5 termina: ora non c'è più posto per caricare P_4 .

gli indirizzi a cui collegare i programmi? Ci sono due modi per risolvere questo problema: fare in modo che sia il caricatore a rilocare il programma (con una tecnica del tutto simile a quella usata dal collegatore) in modo da adattarlo all'indirizzo di caricamento; oppure, compilare tutti i programmi in modo che siano indipendenti dalla posizione.

C'è però un secondo svantaggio, molto più grave: che succede se un processo viene rimosso dalla memoria e poi caricato in una posizione diversa, come il processo P_2 in Figura 3(c)? In generale non funzionerà, in quanto il suo stato potrebbe ora contenere indirizzi che erano validi solo nella posizione originaria: si pensi agli indirizzi di ritorno salvati in pila, o ai puntatori al prossimo elemento scritti in qualche lista. Questi indirizzi andrebbero corretti in base alla nuova posizione, ma in questa architettura è praticamente impossibile trovarli: lo stato è una sequenza di byte e non c'è niente che permetta di distinguere un indirizzo da qualunque altra cosa.

Per rimediare a questo problema modifichiamo il comportamento del registro LINF. Facciamo in modo che, ad ogni accesso in memoria eseguito da livello utente, la CPU *sommi* il contenuto di LINF all'indirizzo generato dal programma, controllando che il risultato non superi LSUP. Il registro LINF contiene dunque una "base" per tutti gli indirizzi generati dal processo. I programmi utente possono ora essere collegati a partire dall'indirizzo zero e i processi che li eseguono possono continuare ad usare gli stessi indirizzi per tutta la loro esecuzione, anche se il loro stato viene tolto dalla memoria e ricaricato in seguito ad un indirizzo diverso. Infatti, siccome LINF conterrà ora il nuovo indirizzo, tutti gli accessi in memoria verranno automaticamente aggiustati in modo da puntare alle locazioni corrette.

Si presti attenzione al fatto che questa correzione è operata a tempo di esecuzione dall'hardware ed è controllata dal software di sistema, l'unico che può scrivere nei registri LINF e LSUP. Gli utenti non possono vedere il meccanismo in opera, né tantomeno modificarlo. Quello che gli utenti vedono è che ora ogni

processo sembra avere una memoria tutta per sé. Questo perché il significato di un indirizzo dipende ora dal contesto: l'indirizzo x di un processo P_1 non è lo stesso indirizzo x di un diverso processo P_2 , in quanto ogni volta che P_1 usa x leggerà o scriverà una locazione di memoria diversa da quella letta o scritta da P_2 , per via dei diversi valori di LINF automaticamente sommati a x . Possiamo dire che ogni processo ha acquisito una sua “memoria virtuale”, che è l'unica a cui ha accesso. Tutti gli indirizzi generati durante l'esecuzione di un processo (sia per prelevare le istruzioni che per leggere o scrivere gli operandi in memoria) sono relativi alla memoria virtuale del processo, e sono detti “indirizzi virtuali”. L'azione di sommare LINF agli indirizzi generati dal processo corrisponde ad una *traduzione* da indirizzi virtuali a indirizzi *fisici*, che possono essere usati per accedere alla memoria del sistema, che da ora in poi chiameremo “memoria fisica”. I processi utente non hanno alcun controllo sulla traduzione, e dunque nessun accesso diretto agli indirizzi fisici: questo ci garantisce che la loro esecuzione non possa dipendere dagli indirizzi fisici, dando la libertà al sistema di allocarli come meglio crede.

1.4 Condivisione e frammentazione

Il meccanismo a cui siamo arrivati ha ancora due svantaggi.

- Nel caso in cui volessimo condividere parte della memoria tra due o più processi (che è proprio il nostro caso), come possiamo fare?
- Che succede nella situazione di Figura 2(d) in cui dobbiamo caricare un processo, ma lo spazio in memoria è frammentato in porzioni troppo piccole?

Il secondo problema si potrebbe risolvere ricompattando lo spazio, ma per farlo dobbiamo copiare la memoria dei processi da una zona all'altra, operazione molto costosa. Entrambi i problemi, invece, potrebbero essere risolti se potessimo “spezzare” la memoria di un processo in più porzioni e gestire ogni porzione separatamente: potremmo dire che alcune porzioni sono condivise e altre no e caricare ogni porzione in uno spazio diverso. In presenza di memoria virtuale, questo “spezzettamento” sarebbe possibile se potessimo tradurre in modi diversi parti diverse dello spazio di indirizzamento di un processo. Questo è ciò che ci permette di fare il meccanismo della *paginazione*.

2 Paginazione

Un meccanismo che ci permette di risolvere tutti i problemi che abbiamo illustrato è quello della *paginazione*. L'idea è di considerare tutti i possibili indirizzi generabili da un processo e di raggrupparli in regioni naturali dette “pagine” e di applicare una traduzione di indirizzi diversa per ogni pagina. In questo modo il sistema è libero di caricare la memoria di un processo ovunque vi sia spazio libero, anche se non contiguo. Inoltre, due o più processi possono condividere

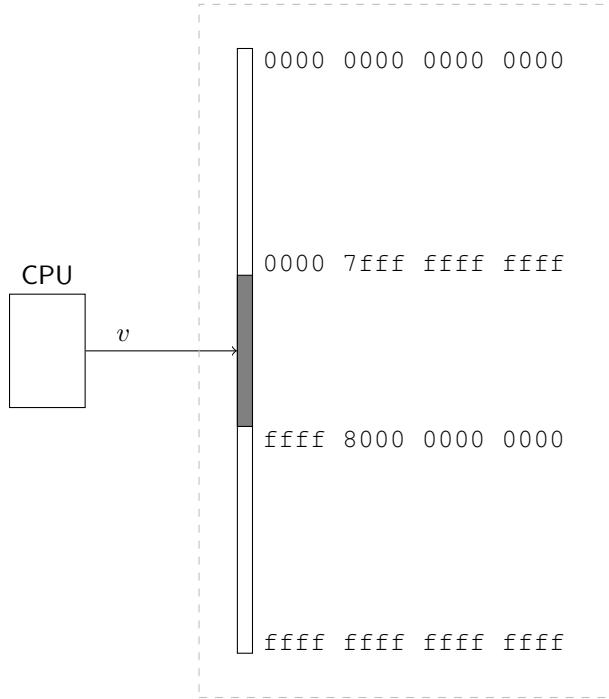


Figura 4: Spazio di indirizzamento nei processori Intel/AMD a 64 bit. Si ricordi che, per queste macchine, solo gli x bit meno significativi di un indirizzo di 64 bit possono assumere un valore qualsiasi, dove x vale 48 o 57 a seconda del modello della CPU. I 64 – x bit più significativi devono essere tutti uguali al bit n. $x - 1$ (contando da 0). Per fissare le idee consideriamo solo il caso $x = 48$. La parte in grigio non è dunque utilizzata. Il processore genera una eccezione se si tenta di usare un indirizzo che rientra in questa parte.

parte della loro memoria: è sufficiente applicare le stesse traduzioni per le pagine che contengono la zona da condividere.

Nel caso dei processori Intel/AMD a 64 bit i possibili indirizzi sono mostrati in Fig. 4, dove si mostra solo il collegamento tra la CPU e la memoria tramite il bus degli indirizzi. L’insieme di questi indirizzi forma lo *spazio di indirizzamento virtuale* di un processo. Gli indirizzi che abbiamo usato fino ad ora (quelli a cui risponde la memoria centrale, la memoria video, l’APIC e eventuali altre periferiche con registri mappati in memoria) fanno invece parte dello *spazio di indirizzamento fisico*. Gli indirizzi che fanno riferimento allo spazio di indirizzamento virtuale sono detti *indirizzi virtuali*, e quelli che fanno riferimento allo spazio di indirizzamento fisico sono detti *indirizzi fisici*. Si noti che non facciamo più riferimento soltanto alla “memoria” intesa come RAM ma parliamo più in generale di “spazio di indirizzamento”, appunto perché, come appena ricordato, anche altre cose possono essere accessibili tramite indirizzi di memoria.

Suddividiamo idealmente lo spazio di indirizzamento virtuale di Figura 4 in regioni naturali¹, che chiamiamo *pagine*. Per realizzare la traduzione suddividiamo anche lo spazio di indirizzamento fisico in regioni naturali, dette *frame* (cornici), della stessa dimensione delle pagine. Assumiamo che le pagine e i frame siano grandi 4 KiB (0x1000 in esadecimale). Ogni indirizzo virtuale può essere scomposto in (p, o) , dove p è il *numero di pagina* e o l'offset all'interno della pagina. Similmente, ogni indirizzo fisico può essere scomposto in (n, q) , dove n è un *numero di frame* e q un offset all'interno del frame.

Il meccanismo della paginazione ci permette di mappare qualunque pagina su qualunque frame. Per farlo si introduce una struttura dati che, dato un numero di pagina p , ci permette di conoscere il numero di frame n su cui p è mappata. La più semplice struttura dati possibile è un array indicizzato dal numero di pagina: se chiamiamo \mathbf{a} questo array, il numero di frame che cerchiamo è $n = \mathbf{a}[p]$. Chiamiamo *tavella di corrispondenza* l'array \mathbf{a} . In Fig. 5 si mostra come la tavella di corrispondenza sia un vettore in cui ogni elemento, in ordine, si occupa della traduzione della corrispondente pagina, in ordine, dello spazio di indirizzamento virtuale. Supponiamo ora che la CPU generi un indirizzo v , per esempio per accedere ad una variabile di un programma caricato in memoria centrale. La cella a cui la CPU vuole accedere non si trova in memoria all'indirizzo v : come nel caso dei registri LINF e LSUP, l'indirizzo v deve essere *tradotto* prima di poter essere usato per accedere alla memoria. La traduzione tramite LINF consisteva semplicemente in una somma, mentre ora è più complessa. L'indirizzo v cadrà all'interno di una certa pagina, sia la numero p , ad un certo offset o all'interno della pagina. Se p è caricata in $n = \mathbf{a}[p]$, la cella che corrisponde all'indirizzo v sarà quella che si trova all'offset o dentro il frame numero n . In altre parole, l'indirizzo (p, o) viene tradotto in $(\mathbf{a}[p], o)$ (si noti che l'offset resta lo stesso).

Per eseguire questa traduzione di indirizzi introduciamo un nuovo dispositivo tra la CPU e la memoria: la Memory Management Unit (MMU). La MMU intercetta tutti gli indirizzi generati dalla CPU e li traduce in base alla tavella di corrispondenza attiva, seguendo il procedimento che abbiamo descritto sopra: sostituire il numero di pagina con il numero di frame letto dalla tavella di corrispondenza, lasciando l'offset inalterato. L'operazione di traduzione è riassunta in Figura 6.

La paginazione ci permette di risolvere quasi tutti i problemi da cui eravamo partiti.

- *Isolamento tra i processi.* Si ottiene utilizzando una diversa tabella di corrispondenza per ogni processo e impedendo che le tabelle di corrispondenza possano essere manipolate da livello utente. In questo modo ogni processo può accedere solo a quelle parti della memoria fisica che si trovano nel codominio della funzione di traduzione realizzata dalla MMU. Per impedire a un processo P di accedere al contenuto di un qualunque frame n è sufficiente che n non compaia mai nella tabella di corrispondenza di P .

¹Si ricordi che abbiamo chiamato “regioni naturali” intervalli di indirizzi allineati naturalmente, con una dimensione che sia una potenza di 2.

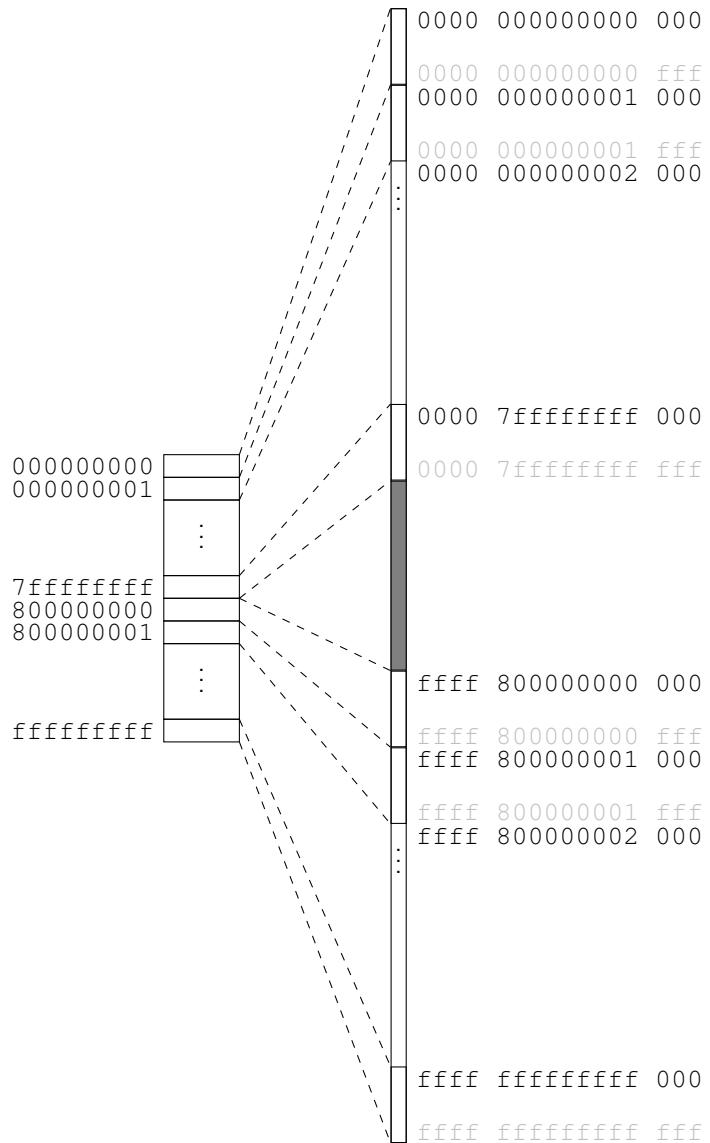


Figura 5: Tabella di corrispondenza e memoria virtuale. La tabella di corrispondenza è sulla sinistra, con a fianco gli indici (in esadecimale) delle sue entrate. Sulla destra è rappresentato lo spazio di indirizzamento virtuale, con gli indirizzi iniziali e finali delle pagine. Tutti gli indirizzi che cadono dentro una pagina sono tradotti usando l'entrata collegata tramite le linee tratteggiate. Si noti che, per il modo in cui sono definiti gli indirizzi possibili nell'architettura Intel/AMD a 64 bit, quando si passa dall'elemento di indice (in base 16) 7fffffff fff a quello di indice 800000000 si salta dalla pagina virtuale di indirizzo 0000 7fffffff 000 a quella di indirizzo ffff 80000000 000.

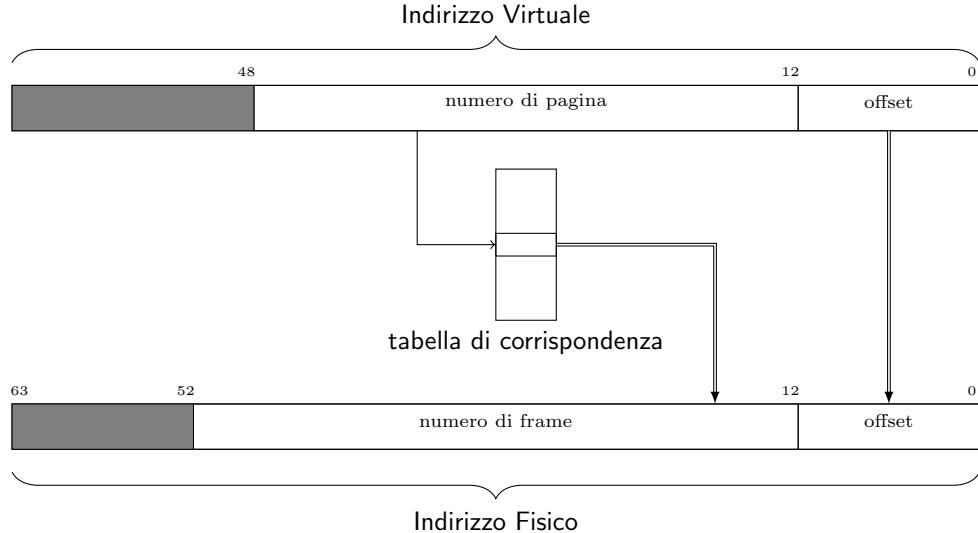


Figura 6: Traduzione da indirizzo virtuale a fisico. Si noti come in questa architettura il numero di pagina è dato solo dai bit 12–47 dell'indirizzo virtuale, mentre il numero di frame è più grande e occupa i bit 12–51 dell'indirizzo fisico.

- *Caricamento a indirizzi variabili.* I processi usano solo ed esclusivamente gli indirizzi virtuali, che non cambiano anche se i processi vengono rimossi dalla memoria e caricati in un altro punto. Inoltre, visto che ogni processo ha la sua memoria virtuale indipendente, gli indirizzi di collegamento possono essere scelti liberamente senza problemi di collisione.
- *Condivisione.* Se si vuole che due o più processi condividano della memoria, è sufficiente che il sistema inserisca gli stessi numeri di frame nelle entrate opportune delle varie tabelle.
- *Frammentazione.* Ogni pagina può essere caricata ovunque ci sia un frame libero, non è più necessario caricare un intero processo in una porzione contigua della memoria.

Il programmatore deve comunque dire al sistema quanto spazio occupa ogni processo, con l'unica differenza che ora lo spazio si misura in pagine.

2.1 Funzioni aggiuntive

La presenza della MMU permette al sistema di realizzare anche altre funzioni. La MMU, infatti, si trova in un punto in cui può osservare tutti gli indirizzi generati dal software e, per ognuno di essi, consulta la tabella di corrispondenza. La tabella può essere usata per associare ulteriori informazioni ad ogni pagina (oltre al numero di frame) che dicano alla MMU come comportarsi quando

intercetta un indirizzo che cade in quella pagina. Le entrate delle tabelle che si trovano nell'architettura AMD/Intel includono i seguenti campi:

- un flag P che dice se la traduzione è valida;
- un flag R/W che dice se sono ammesse scritture nella pagina;
- un flag U/S che dice se sono ammessi accessi alla pagina da livello utente.
- due flag, PWT e PCD, per agire sulla cache.
- due flag, A e D, che danno indicazioni sugli accessi che la MMU ha osservato sull'pagina.

Il flag P serve a marcare le pagine che il processo non usa. Si ricordi che la tabella di corrispondenza contiene una entrata per ogni possibile pagina. Il caricatore di sistema (nel nostro caso, la `activate_p()`), provvederà a porre P=0 in tutte le pagine di cui il processo non ha bisogno. Se la MMU riceve dalla CPU un indirizzo che porta ad una entrata con P=0, fa in modo che la CPU sollevi una eccezione. Il sistema, tipicamente, terminerà il processo con un errore². Il bit P può essere anche utilizzato per fermare i processi quando cercano di dereferenziare un puntatore nullo: è sufficiente porre P=0 nell'entra di indice 0 (relativa alla pagina numero 0).

Il flag R/W può essere usato per proteggere dalla scrittura la sezione `.text`. Infatti, anche se l'architettura detta di “von Neumann” era stata introdotta proprio per permettere ai programmi di modificare il proprio stesso codice, questa pratica è da tempo fortemente limitata, in quanto crea molti più problemi di quanti ne risolva³. Tipicamente viene completamente vietata per i processi utente, settando opportunamente i flag R/W. La MMU fa sollevare una eccezione anche quando incontra R/W=0 nel tradurre l'indirizzo durante una operazione di scrittura.

Per capire la necessità del flag U/S, consideriamo che la MMU presente nei sistemi Intel/AMD è *sempre* attiva, anche quando il processore si trova a livello sistema. Se vogliamo che sia possibile saltare al codice di sistema quando un processo invoca una primitiva, genera una eccezione o riceve una interruzione esterna, dobbiamo fare in modo che tutta la memoria M1 si trovi nel codominio delle funzioni di traduzione di tutti i processi. Dobbiamo dunque riservare delle pagine nella memoria virtuale di ogni processo, in modo che vengano tradotte nei frame che coprono M1. Nel processore AMD64 sfruttiamo il fatto che la memoria virtuale è naturalmente divisa in due (Figura 4) e riserviamo la parte superiore al sistema e la parte inferiore all'utente⁴. Allo stesso tempo non vogliamo che i processi utente possano accedere alla memoria M1. Potremmo

²In Unix il processo riceve una signal `SIGSEV`, che normalmente ne causa la terminazione con ritorno alla shell, che poi stampa “Segmentation fault”.

³Si noti che l'articolo originale di von Neumann già limitava i modi in cui i programmi dovevano poter modificare il proprio codice, ma i primi calcolatori poi realizzati non applicavano queste limitazioni. In ogni caso, anche il meccanismo più limitato suggerito da von Neumann non è realmente necessario.

⁴Linux e FreeBSD fanno al contrario.

ricorrere nuovamente al registro limite, ma ora che abbiamo la MMU possiamo sfruttarla per farle fare anche questo controllo: basta porre U/S=sistema in tutte le entrate relative alla parte alta dello spazio di indirizzamento (le entrate con indici da 00000000 a 7fffffff in Figura 5). Anche in questo caso la MMU fa generare una eccezione se rileva un accesso da livello utente ad una pagina con U/S=sistema.

I bit PWT e PCD sono un mezzo tramite il quale il sistema può indirettamente inviare ordini alla cache, sfruttando il fatto che la MMU si trova sul percorso che porta dal processore alla cache (la cache si trova tra la MMU e la memoria fisica). La MMU si preoccuperà di inoltrare l'ordine alla cache ogni volta che traduce un indirizzo.

1. Il bit PCD (Page Cache Disable) ordina alla cache di non intercettare l'operazione (sia essa di lettura o di scrittura) e di lasciarla passare inalterata sul bus, similmente a come si comporta per gli accessi nello spazio di I/O. La routine di inizializzazione porrà PCD=1 per tutte le pagine che contengono indirizzi di registri di I/O mappati in memoria, invece che locazioni di memoria. Un esempio è l'APIC, i cui indirizzi sono appunto mappati nello spazio di memoria.
2. Il bit PWT (Page Write Through) ordina alla cache di usare la politica *write-through* per questo particolare accesso (ovviamente, solo se si tratta di una scrittura). Se PCD è 1 il bit PWT non ha effetto. Se PCD è 0, porre PWT=1 può essere utile per la parte di indirizzi relativa alla memoria video: quando il programma scrive vogliamo che la scrittura arrivi nella vera memoria video e non si fermi in cache, in modo che il controllore video possa visualizzare l'informazione sul display; ma se il programma vuole leggere dalla memoria video possiamo tranquillamente farlo leggere dalla cache.

Incidentalmente, si noti che l'aver posizionato la cache dopo la MMU comporta che la cache non deve subire modifiche rispetto a quella che abbiamo già studiato: il controllore cache vede arrivare indirizzi fisici esattamente come prima (arrivano dalla MMU invece che direttamente dalla CPU, ma questo è irrilevante). Inoltre, la presenza della cache non turba il funzionamento della MMU: la presenza della cache è trasparente anche per la MMU. Questa è la soluzione adottata nei processori Intel/AMD64.

I bit A e D, infine, sono legati all'implementazione della *paginazione su domanda*, che non tratteremo. La MMU setta ad 1 il bit A di una entrata quando la usa, cioè durante un accesso ad un indirizzo all'interno della corrispondente pagina. Se l'accesso era in scrittura, la MMU setta anche il bit D.

3 La Super-MMU

Precedentemente, per fare in modo che ogni processo fosse isolato dagli altri, avevamo una coppia di valori `contesto[I_LINF]` e `contesto[I_LSUP]` per

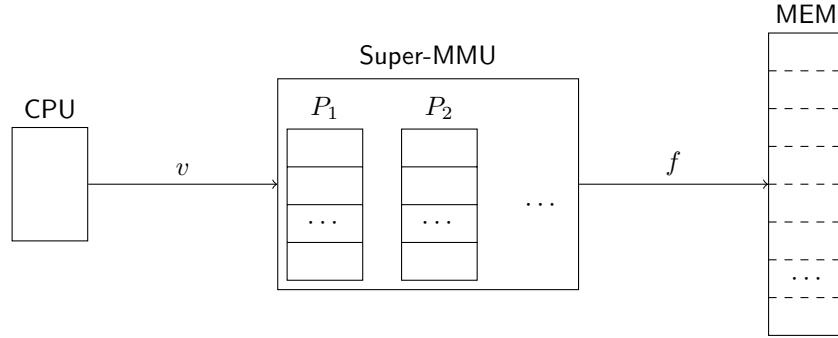


Figura 7: Traduzione degli indirizzi con una Super-MMU. Quando è in esecuzione il processo P_i , la Super-MMU traduce gli indirizzi usando la corrispondente tabella.

ogni processo, con una sola coppia attiva ad ogni istante (quella appartenente al processo in esecuzione). Analogamente, ora dovremo avere una intera tabella di corrispondenza distinta per ogni processo, con una sola tabella attiva ad ogni istante (quella appartenente al processo in esecuzione). Per introdurre i dettagli un po' alla volta, in modo da concentrarci prima sugli aspetti più importanti, introduciamo prima una Super-MMU che contiene essa stessa tutte le tabelle di corrispondenza, di tutti i processi, al proprio interno (Figura 7).

È disponibile un semplice esempio che illustra quanto detto finora, facendo uso della Super-MMU. Uno degli scopi dell'esempio è di far vedere come la memoria virtuale sia completamente *trasparente* al programmatore utente.

Tabelle multilivello

G. Lettieri

15 Maggio 2021

1 MMU₁: tabella su più livelli

Proviamo a calcolare quanto sono grandi le tabelle di corrispondenza usate dalla Super-MMU. La memoria virtuale è almeno di 2^{48} byte¹ e ogni pagina è grande 2^{12} byte, quindi la memoria virtuale contiene

$$\frac{2^{48}}{2^{12}} = 2^{36} = 64 \text{ Gi pagine.}$$

La tabella di corrispondenza di ogni processo deve avere una entrata per ognuna di queste pagine. Ogni entrata deve contenere almeno i bit P, R/W, U/S, PCD, PWT, A, D e il numero di frame che fornisce i bit da 12 a 51 dell'indirizzo fisico, per un totale di 43 bit, arrotondati in 6 byte. Se poi vogliamo che la dimensione di ogni entrata sia una potenza di 2 dovremo usare almeno 8 byte. In conclusione, la tabella di corrispondenza dovrà essere di

$$64 \text{ Gi} \times 8 \text{ B} = 512 \text{ GiB.}$$

Difficilmente, quindi, possiamo pensare di avere un dispositivo di memoria che possa contenere anche una sola di queste tabelle.

Per affrontare il problema notiamo che la stragrande maggioranza dei programmi ha bisogno soltanto di una piccola frazione dei 2^{48} byte disponibili di memoria virtuale. Vorremmo avere una struttura dati che contenga le sole entrate effettivamente utilizzate.

Introduciamo quindi MMU₁, una MMU del tutto identica alla Super-MMU, tranne che per il formato della tabella di corrispondenza. Come la Super-MMU, MMU₁ possiede una memoria interna in cui salvare le tabelle di corrispondenza e un registro, **cr3**, che serve ad individuare la tabella di corrispondenza attiva ad ogni istante. La speranza è di poter usare una memoria interna molto più piccola rispetto a quella richiesta dalla Super-MMU.

¹Come detto precedentemente, omettiamo di considerare il caso di memoria virtuale grande 2^{57} byte.

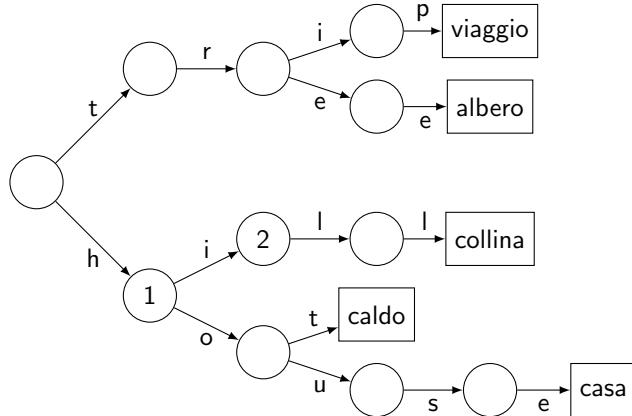


Figura 1: Un esempio di trie con chiavi e valori di tipo stringa.

1.1 La struttura dati *trie*

La struttura dati utilizzata da MMU₁ è un *bitwise trie*, che è una variante di trie. I trie sono strutture dati ad albero che permettono di mappare chiavi di tipo stringa in valori, in modo che i caratteri successivi della chiave guidino la ricerca all'interno dell'albero. Consideriamo, per esempio, il trie mostrato in Figura 1. L'albero memorizza le associazioni *trip* \mapsto *viaggio*, *tree* \mapsto *albero*, *hill* \mapsto *collina*, *hot* \mapsto *caldo* e *house* \mapsto *casa*. Si noti come gli archi dell'albero siamo marcati con i caratteri delle chiavi e il valore associato ad ogni chiave si trovi nella foglia che si raggiunge partendo dalla radice e seguendo il percorso indicato dalla chiave.

Un modo di implementare un trie è di avere, in ogni nodo dell'albero, un array di 128 entrate, ciascuna delle quali contenga il puntatore al prossimo nodo da visitare in base al codice ASCII del prossimo carattere della chiave.

Ogni nodo si trova sul percorso di tutte le chiavi che iniziano con lo stesso prefisso. Per esempio, il nodo marcato con “2” si trova nel percorso di tutte le chiavi che iniziano con “hi”. Un puntatore nullo nell’array di un nodo indica che il trie non contiene chiavi con il corrispondente prefisso. Per esempio, una ricerca della chiave “history” nel trie di Figura 1 seguirebbe il ramo “h” dalla radice per arrivare al nodo “1”, quindi il ramo “i” per arrivare al nodo “2”. Qui troverebbe un puntatore nullo associato al carattere “s” e la ricerca si concluderebbe con un fallimento. L’inserimento di una nuova associazione chiave/valore nel trie comporta una visita dell’albero come in una ricerca, ma creando eventuali nodi mancanti fino alla foglia che deve contenere il valore.

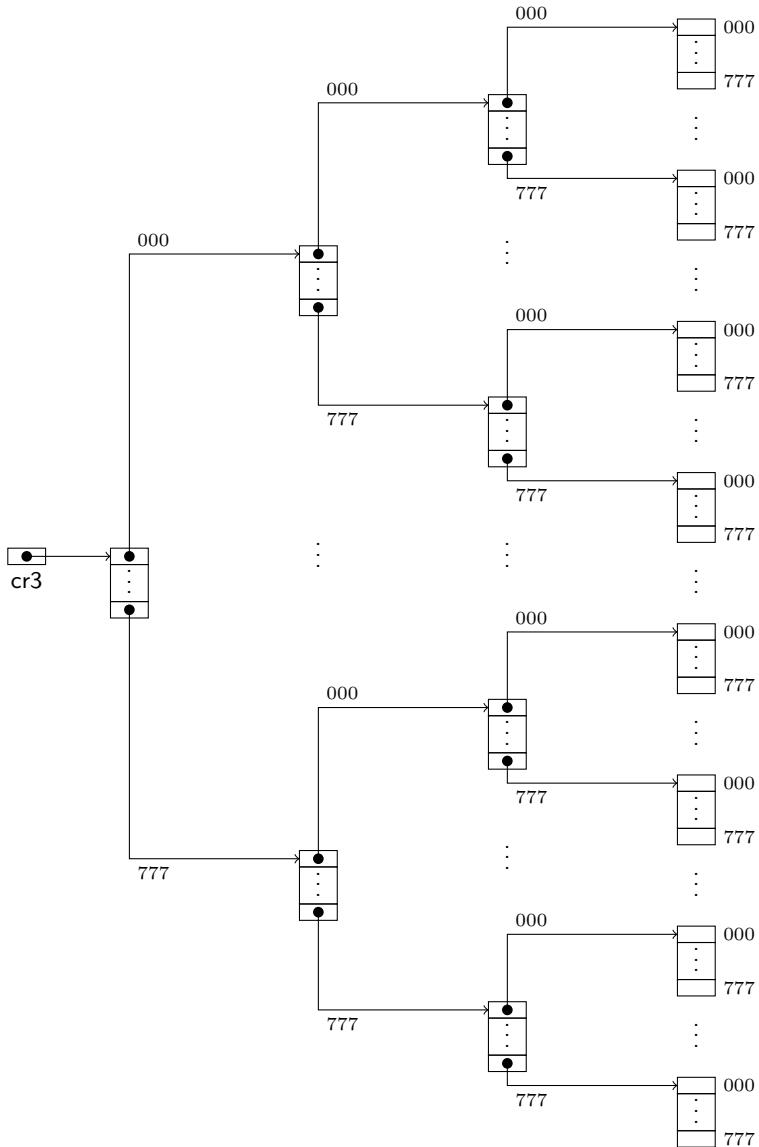
Nel nostro caso la chiave è il numero di pagina e il valore che vi vogliamo associare è il corrispondente numero di frame. Per questo scopo possiamo usare un *bitwise trie*, che funziona esattamente come un trie ma, al posto dei caratteri, usa gruppi di bit della chiave. In particolare, il numero di pagina è composto da 36 bit che possiamo raggruppare in 4 gruppi di 9 bit. Ogni nodo del bitwise trie

conterrà dunque una tabella di $2^9 = 512$ entrate con puntatori, eventualmente nulli, al nodo successivo. Le foglie stesse possono essere tabelle indicizzate dall'ultimo gruppo di 9 bit della chiave. In questo caso le entrate delle tabelle foglie conterranno il numero di frame associato al numero di pagina. Si arriva dunque alla struttura dati illustrata in Figura 2. Ciascun nodo dell'albero di Figura 2, foglie incluse, è una tabella di 512 entrate, ciascuna grande 8 byte. Ogni tabella è grande dunque 4096 byte. L'albero ha al massimo 4 livelli, che per convenzione vengono numerati da 4 a 1 (il livello delle foglie), in modo da poter parlare di tabelle di livello 4, di livello 3 e così via. È molto comodo rappresentare i numeri di pagina in base 8, in quanto ciascuna cifra in base 8 corrisponde a 3 bit del numero di pagina, e dunque ciascun gruppo di 9 bit può essere rappresentato da 3 cifre in base 8. Per esempio, supponiamo che la MMU₁ si trovi a dover tradurre l'indirizzo virtuale $v = (000\ 777\ 000\ 777\ 1234)_8$. Il numero di pagina è $(000\ 777\ 000\ 777)_8$. I primi 9 bit del numero di pagina sono $(000)_8$. La MMU₁ usa quindi l'entrata di indice 0 (vale a dire la prima entrata) della tabella di livello 4 per trovare la prossima tabella da consultare. La MMU₁ passerà dunque alla tabella di livello 3 in alto in Figura 2. Qui usa i successivi 9 bit, $(777)_8$ per sapere quale entrata di questa tabella deve consultare per raggiungere la prossima tabella, di livello 2. Dunque la MMU₁ userà l'ultima entrata della tabella e passerà alla seconda tabella di livello 2 dall'alto in Figura 2. Qui userà i bit $(000)_8$ del terzo gruppo e proseguirà verso la terza tabella di livello 1 dall'alto. Infine, troverà la traduzione che stava cercando nell'entrata $(777)_8$ di questa tabella.

Il formato delle entrate delle tabelle di livello 1 è mostrato in Fig. 3 e rispecchia quello dell'architettura Intel/AMD a 64 bit. Chiameremo queste entrate *descrittori di pagina virtuale* o anche *descrittori di livello 1*, essendo contenuti nelle tabelle di livello 1. Si noti che i descrittori contengono tutte le informazioni che abbiamo già introdotto parlando della Super-MMU. I bit P, R/W, U/S, PWT e PCD sono scritti dalla routine di sistema che attiva il processo (nel nostro caso, `activate_p()`) e soltanto letti dalla MMU₁. I bit A e D, invece, sono letti e scritti sia dal software (di sistema) che dalla MMU₁.

Ogni volta che il sistema carica le pagine di un processo in memoria (alla prima attivazione, oppure dopo uno *swap-in* in seguito ad uno *swap-out*), dovrebbe porre D=0 in tutte le entrate della tabella di corrispondenza. Al momento di eseguire uno *swap-out* del processo, il sistema può evitare di salvare tutte le pagine del processo nel dispositivo di swap ri-esaminando le entrate e notando in quali di esse il bit D è diventato 1: queste puntano a pagine che sono state modificate e vanno risalvate; per tutte le altre il salvataggio si può evitare, in quanto la copia già presente nello swap è ancora valida. Il bit A può essere usato per capire quali pagine/tabelle sono più usate o sono state usate più recentemente, ed è di ausilio soprattutto all'implementazione della paginazione su domanda.

Tutte le tabelle di livello 2, 3 e 4 hanno lo stesso formato. Ciascuna di esse contiene 512 entrate con il formato illustrato in Fig. 4. Il formato è simile a quello dei descrittori di livello 1 mostrati in Fig. 3: ci sono ancora i bit P, R/W e U/S e A, nelle stesse posizioni dei bit omonimi di Fig. 3; il campo “Indirizzo



Liv. 4	Liv. 3	Liv. 2	Liv. 1	Num. tabelle
1	512	256 Ki	128 Mi	Dim. tot.
4 KiB	2 MiB	1 GiB	512 GiB	

Figura 2: Tabella di corrispondenza su 4 livelli, implementata con un bitwise trie.

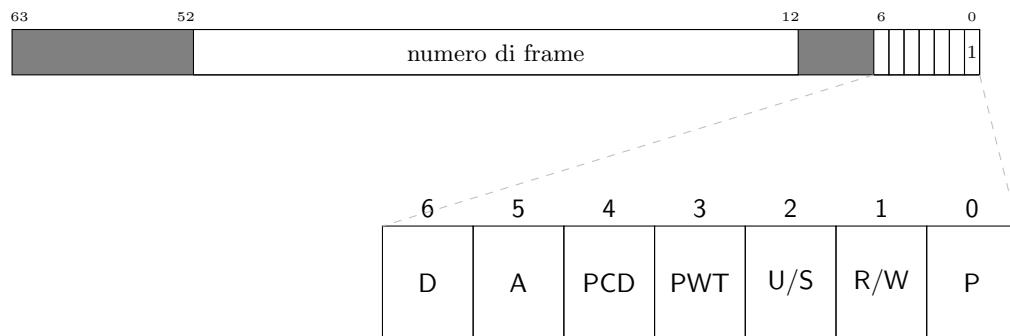


Figura 3: Descrittore di pagina virtuale (tabelle di livello 1).

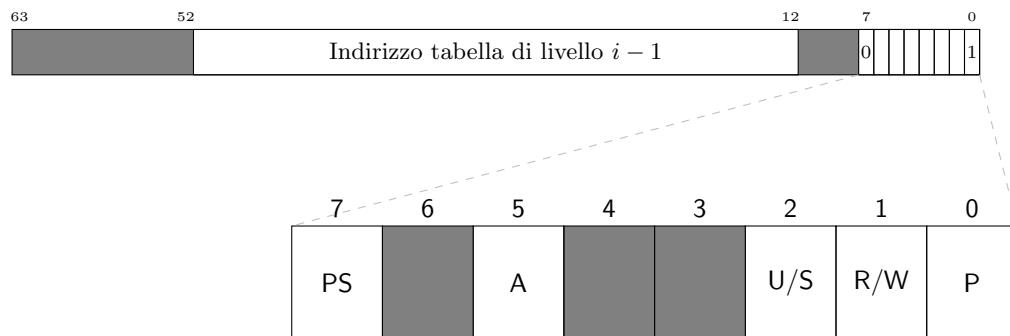


Figura 4: Descrittore di livello i , con $i = 2, 3, 4$.

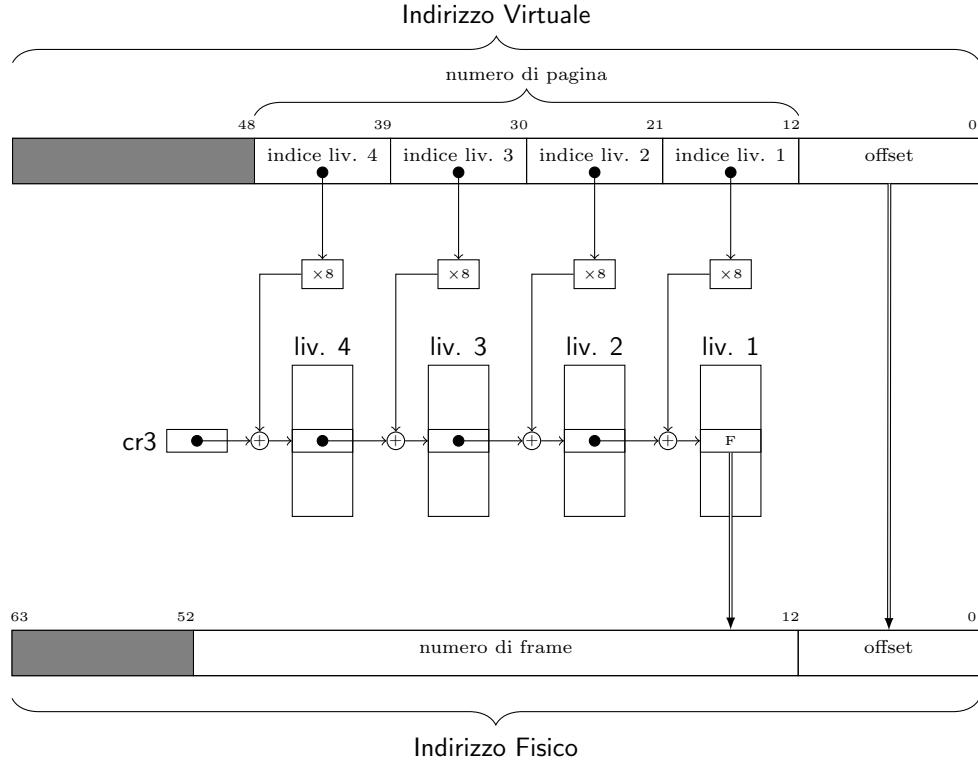


Figura 5: Traduzione da indirizzo virtuale a fisico (pagine di 4 KiB).

tabella di livello $i - 1$ ” occupa la stessa posizione del campo “numero di frame” di Fig. 3 per il momento non ci preoccupiamo del nuovo bit PS e assumiamo che valga 0. Si noti che all’indirizzo della tabella mancano i bit da 0 a 11: la MMU₁ assume che questi bit siano 0, cioè che tutte le tabelle partano da indirizzi che sono multipli di 4 KiB (allineamento naturale). Questo vale anche per la tabella di livello 4: i 12 bit meno significativi di **cr3** devono essere tutti a 0.

Chiameremo le entrate delle tabelle di livello 2 *descrittori di livello 2* o *descrittori delle tabelle di livello 1*. Si noti il decremento del livello: i descrittori di livello 2 sono contenuti in tabelle di livello 2 e descrivono tabelle di livello 1. Allo stesso modo chiameremo le entrate delle tabelle di livello 3 *descrittori di livello 3* o *descrittori delle tabelle di livello 2* e, infine, chiameremo le entrate della tabella di livello 4 *descrittori di livello 4* o *descrittori delle tabelle di livello 3*.

Anche se simili, i descrittori di livello 2, 3 e 4 non vanno confusi con i descrittori di livello 1: i primi descrivono tabelle, mentre quelli di livello 1 descrivono pagine. I descrittori di tabella servono a trovare le tabelle di livello inferiore, ma solo le tabelle foglia contengono la traduzione cercata.

In Fig. 5 abbiamo riassunto il processo di traduzione operato dalla MMU₁,

mostrando più in dettaglio le operazioni svolte da MMU₁, assumendo che tutti i bit P valgano 1. Al primo passo MMU₁ deve leggere il corretto descrittore di livello 4, che si trova nella sua memoria. Per farlo deve eseguire una operazione di lettura in memoria, ovviamente specificando l'indirizzo. La tabella di livello 4 è un vettore di descrittori, ciascuno grande 8 byte, e la MMU₂ vuole leggere il descrittore il cui indice è contenuto nei bit 39–47 di V (indice di livello 4), sia i_4 . L'indirizzo a cui vuole leggere è dunque $\text{cr}3 + i_4 \times 8$. Si noti che, per quanto detto sull'allineamento naturale delle tabelle, questa operazione non comporta una vera somma, ma solo una concatenazione di bit. Anche la moltiplicazione per 8 consiste, ovviamente, nella concatenazione con tre bit costanti pari a 0. Letto il descrittore di livello 4, MMU₂ ne estrae il campo indirizzo (bit 12–51), lo concatena con i bit 30–38 di V e con altri tre bit a 0, ottenendo così l'indirizzo del descrittore di livello 3. E così anche per i descrittori di livello 2 e 1. Arrivata al livello 1, la MMU₂ estrae il campo F (bit 12–51), lo concatena con l'offset di V e ottiene così la traduzione.

Durante la traduzione la MMU₁ esegue anche altri compiti, analoghi ai compiti aggiuntivi che svolgeva la Super-MMU, ma applicati alla struttura multi-livello:

- controlla tutti i bit R/W: una operazione di scrittura è permessa solo se tutti e 4 i bit lungo il percorso la permettono;
- controlla tutti i bit U/S: una operazione (di lettura o scrittura) è permessa solo se tutti e 4 i bit lungo il percorso la permettono;
- passa al controllore cache le informazioni contenute nei bit PWD e PCD nel descrittore di livello 1;
- pone a 1 tutti e 4 i bit A incontrati, se non lo erano già;
- in caso di scrittura, pone a 1 il bit D nella tabella di livello 1 (i descrittori di livello maggiore di 1 non hanno il bit D).

Se uno qualunque dei bit P incontrati durante la traduzione vale 0, la MMU₁ smette di tradurre e solleva una eccezione di page fault. La routine di sistema che gestisce il fault terminerà il processo con un errore.

Ciascuna delle tabelle di corrispondenza dalla Super-MMU deve essere sostituita da uno di questi alberi (in altre parole, ci serve un albero per ogni processo). Nella parte bassa di Figura 2 abbiamo riportato il numero massimo di tabelle per ogni livello del trie e quanto spazio occupa ogni livello. Si noti che le tabelle di livello 1 occupano complessivamente 512 GiB, esattamente come la tabella della Super MMU che stiamo cercando di sostituire. In effetti, se rimettessimo insieme tutte le tabelle di livello 1, in ordine, riporteremmo la tabella di corrispondenza della Super-MMU. Siccome a questo spazio dobbiamo aggiungere quello richiesto dai livelli superiori dell'albero, ci rendiamo conto che il trie completo occupa *più* spazio della tabella di corrispondenza originaria. Anche dal punto di vista del tempo richiesto per eseguire la traduzione ci stiamo perdendo: la tabella unica della Super-MMU richiede un unico accesso in memoria,

mentre il trie ne richiede 4. Qual è dunque il vantaggio di passare al trie? Un vantaggio è che possiamo evitare di allocare le parti di trie relative a indirizzi virtuali che il processo non usa. Per esempio, se un processo non usa nessun indirizzo il cui numero di pagina inizi con $(777)_8$, il trie di questo processo non ha bisogno di tutto il sottoalbero inferiore di Figura 2. L'omissione di questo sottoalbero si ottiene semplicemente ponendo a 0 il bit P dell'entrata $(777)_8$ della tabella di livello 4. Dal momento che la maggior parte dei processi userà soltanto una piccola parte dei possibili indirizzi virtuali, ci rendiamo conto che in questo modo il trie avrà quasi sempre una dimensione ragionevole.

1.2 Regioni e sottoregioni

Un altro modo per pensare alle operazioni svolte dalla MMU₁ è di ragionare in termini di regioni naturali (che, ricordiamo, sono intervalli di indirizzi con dimensione pari ad una potenza di 2 e allineate naturalmente). Possiamo identificare ciascuna tabella del trie specificando la sequenza di bit della chiave che porta dalla radice alla tabella in questione. Per esempio, la terza tabella di livello due dall'alto in Figura 2 è identificata dalla sequenza di 18 bit $(777\ 000)_8$. La traduzione di tutti gli indirizzi virtuali che iniziano con questo prefisso deve passare da questa tabella. Questa tabella, dunque, è “responsabile” della traduzione dell'intera regione naturale, grande $2^{48-18} = 2^{30} = 1 \text{ GiB}$, il cui numero di regione è appunto $(777\ 000)_8$. Aggiungendo ulteriori 9 bit possiamo identificare anche ogni singola *entrata* della tabella. Per esempio, i 27 bit $(777\ 000\ 777)_8$ identificano sia la terza tabella di livello 1 dal basso di Figura 2, chiamiamola *t*, sia l'ultima entrata della seconda tabella di livello 2 dal basso, chiamiamola *e*. Di nuovo, la traduzione di tutti gli indirizzi virtuali che iniziano con $(777\ 000\ 777)_8$ deve passare dall'entrata *e* e poi da una delle entrate della tabella *t*. Tutti questi indirizzi virtuali appartengono alla stessa regione naturale grande $2^{48-27} = 2^{21} = 2 \text{ MiB}$, il cui numero di regione è $(777\ 000\ 777)_8$. Possiamo dire che l'entrata *e*, o l'intera tabella *t*, sono responsabili della traduzione in questa regione.

In generale, diremo che ogni entrata di una tabella di livello *i*, con $1 \leq i \leq 4$, sarà responsabile della traduzione di una regione naturale di livello *i* – 1. Invece ogni tabella di livello *i*, nella sua interezza, sarà responsabile della traduzione di una regione naturale dello stesso livello *i*. Le regioni di livello 0 non sono altro che le pagine, grandi 2^{12} byte. In generale una regione di livello *j*, con $0 \leq j \leq 4$, è grande 2^{9j+12} byte. Quindi, ogni entrata di una tabella di livello 1 è responsabile della traduzione di una ben precisa pagina, mentre una intera tabella di livello 1, nel suo complesso, è responsabile della traduzione di una ben precisa regione di livello 1, grande $2^{9 \times 1 + 12} \text{ B} = 2 \text{ MiB}$, la stessa regione di cui è responsabile l'entrata (in una tabella di livello 2) che punta alla tabella nel trie. E così via.

Paginazione: complementi

G. Lettieri

28 Aprile 2022

Eliminiamo ora tutte le semplificazioni che abbiamo introdotto e studiamo la MMU che si trova nei sistemi Intel/AMD a 64 bit.

1 Memoria fisica in memoria virtuale

La MMU₁ aveva una memoria interna per memorizzare le tabelle dei vari livelli, ma per la vera MMU non è così. Le tabelle devono essere memorizzate nella memoria fisica, insieme alle pagine dei processi e al codice e alle altre strutture dati del sistema. Ogni volta che il sistema carica un processo dallo swap deve anche trovare lo spazio per le necessarie tabelle di corrispondenza. Ogni volta che il sistema mette in esecuzione un processo, deve scrivere in `cr3` il numero del frame che contiene la tabella di livello 4 di quel processo.

Concettualmente tutte le tabelle di traduzione farebbero parte di M1: sono strutture dati del sistema, inaccessibili agli utenti. Devono però essere allocate dinamicamente, in base ai processi che vengono creati dalle applicazioni degli utenti. Dal momento che tutte le tabelle sono grandi 4 KiB, come le pagine, conviene allocarle dentro i frame della memoria M2, in modo che tutto lo spazio M2 sia potenzialmente disponibile sia per le pagine, sia per le tabelle, senza dover stabilire *a-priori* quanto spazio dedicare alle une o alle altre. Allocandole nei frame di M2 rispettiamo anche automaticamente i requisiti di allineamento (ricordiamo che tutte le tabelle devono essere allineate naturalmente).

Questo comporta, però, che il sistema deve poter accedere liberamente a tutti i frame di M2, in modo da poter consultare liberamente le tabelle (per esempio, quando deve eseguire lo swap-out di un processo, per andare a ritrovare tutte le pagine). Il sistema deve anche, ovviamente, accedere a tutto M1, quindi dobbiamo permettergli di accedere a tutta la memoria fisica. Allo stesso tempo dobbiamo continuare a negare questo accesso ai processi utente, se non per le parti che contengono le loro pagine. La cosa più semplice sarebbe di avere una MMU che si disattiva ogni volta che la CPU passa a livello sistema, ma la MMU che abbiamo non si comporta così. Possiamo però creare una traduzione che non abbia alcun effetto, che di fatto è equivalente a disattivarla. Creiamo quindi delle traduzioni “identità” che lascino inalterati (li traducano in sé stessi) tutti gli indirizzi che vanno da 0 fino all’ultimo indirizzo della memoria fisica. Poi, inseriamo queste traduzioni nello spazio di indirizzamento di ogni processo.

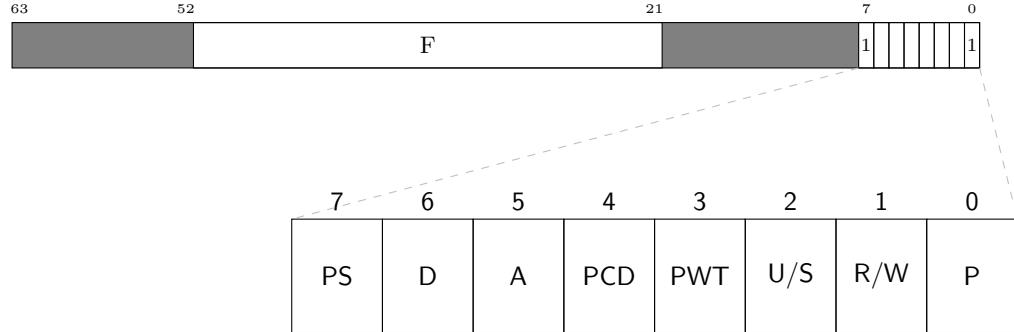


Figura 1: Descrittore di pagina virtuale da 2 MiB (tabelle di livello 2).

Questo permette alle routine di sistema di usare indirizzi fisici proprio come se la MMU fosse disattivata, indipendentemente da quale processo si trova in esecuzione. È come se nella parte alta dello spazio di indirizzamento di ogni processo avessimo creato una finestra che permette di vedere la memoria fisica così come è. Ovviamente questa finestra è accessibile solo da livello sistema, in quanto per tutte le traduzioni nella metà alta dello spazio di indirizzamento abbiamo posto U/S=sistema.

Questa finestra deve essere creata prima di attivare la memoria virtuale. All'avvio del sistema, il processore parte con la memoria virtuale disattivata ed esegue la routine di inizializzazione. Questa può allocare e inizializzare tutte le tabelle necessarie alla definizione della finestra e poi attivare la memoria virtuale. A questo punto la routine di inizializzazione può continuare ad utilizzare indirizzi fisici come stava facendo prima dell'attivazione. Quando si passa a livello utente queste traduzioni diventano inaccessibili e ridiventano accessibili ogni volta che si ritorna a livello sistema.

Le rimanenti caratteristiche della MMU ci permettono di ridurre l'occupazione di spazio e il tempo richiesto per effettuare le traduzioni.

2 Pagine di grandi dimensioni

L'architettura Intel/AMD a 64 bit ci permette di avere pagine più grandi di 4 KiB usando il bit PS nei descrittori di livello 3 e 2. Anche questo bit è scritto dalle routine di sistema e è soltanto letto dalla MMU. La Fig. 1 mostra il formato del descrittore di livello 2 nel caso in cui il bit PS valga 1. Si vede che il descrittore contiene ora un campo F che va dai bit 21 a 51, invece del puntatore alla tabella di livello 1. La Fig. 2 mostra la traduzione da indirizzo virtuale a fisico eseguita in questo caso: quando la MMU arriva alla tabella di livello 2 e trova il bit PS a 1, usa come offset tutti i bit da 0 a 20 e li concatena al campo F trovato nel descrittore. In questo modo abbiamo eliminato una tabella di livello 1, risparmiando il relativo spazio. Il meccanismo è compatibile e può convivere con le pagine di 4 KiB: la MMU si comporta come sempre nei

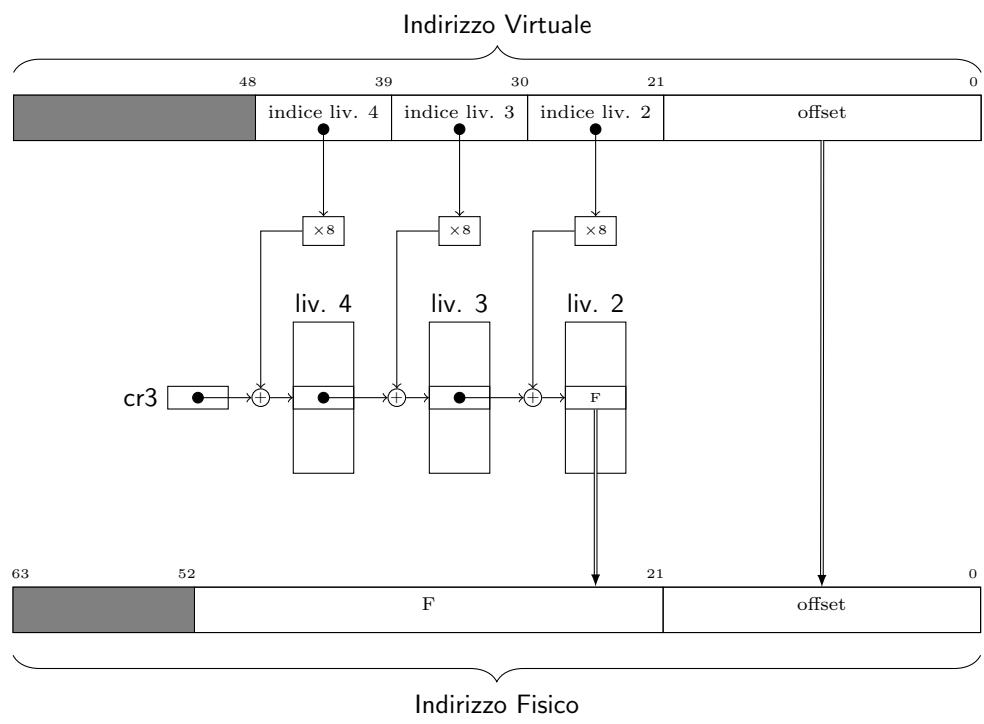


Figura 2: Traduzione da indirizzo virtuale a fisico (pagine di 2 MiB).

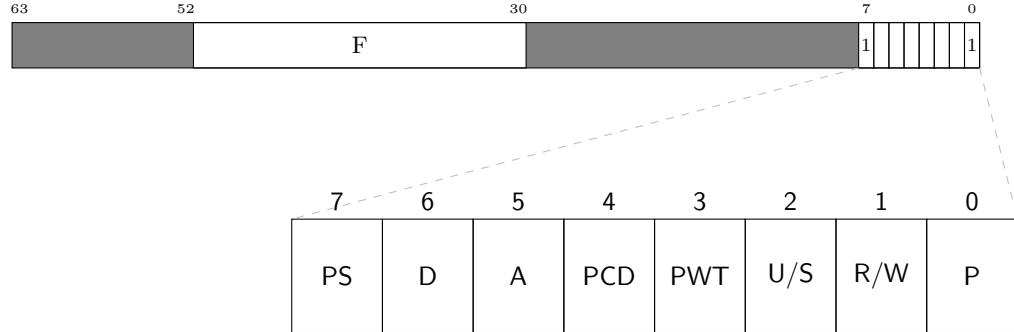


Figura 3: Descrittore di pagina virtuale da 1 GiB (tabelle di livello 3).

livelli 4 e 3 e scopre di dover eseguire il nuovo tipo di traduzione solo quando arriva al livello 2 e trova il bit PS pari a 1; se lo avesse trovato a 0 avrebbe proseguito fino al livello 1, come sempre. Ogni descrittore di livello 2 può avere il bit PS a 1 o a 0 indipendentemente dagli altri, quindi nello stesso spazio di indirizzamento possiamo usare sia pagine di 2 MiB, sia pagine di 4 KiB, a seconda della convenienza.

I processori più recenti permettono di avere PS=1 anche nei descrittori di livello 3. La Fig. 3 mostra il formato del descrittore di livello 3 in questo caso, in cui si vede che il campo F sostituisce anche qui il campo che punta alla tabella di livello 2. La Fig. 4 mostra la traduzione eseguita dalla MMU in questo caso. Si vede come l'offset è ora su 30 bit, e dunque le pagine sono ora grandi 1 GiB. Questo tipo di traduzione è molto utile per creare la finestra sulla memoria fisica (si veda la sezione precedente).

3 Il TLB

Per ogni accesso in memoria la MMU deve prima consultare fino a 4 tabelle per poter eseguire la traduzione¹. Per ogni tabella consultata, se il corrispondente bit A è a zero, la MMU deve anche eseguire un ulteriore accesso (in scrittura) per settarlo a 1. Se consideriamo che il nostro programma deve accedere continuamente in memoria, sia per prelevare le sue stesse istruzioni, sia per prelevare o scrivere gli operandi che si trovano in memoria, ci rendiamo subito conto che l'impatto della MMU sul tempo di esecuzione di un programma può essere molto grande. È vero che subito dopo la MMU c'è la cache, e quindi possiamo sperare che molti di questi accessi non debbano realmente arrivare fino alla memoria, ma resta il fatto che, anche nel migliore dei casi, quello che prima era un unico accesso in cache si è ora trasformato in una sequenza di 5 accessi in cache.

Per affrontare questo problema si introduce una nuova cache, chiamata TLB (Translation Lookaside Buffer), che è specifica per la MMU. Lo scopo di questa

¹Fino a 5 nei modelli con 57 bit significativi nell'indirizzo virtuale.

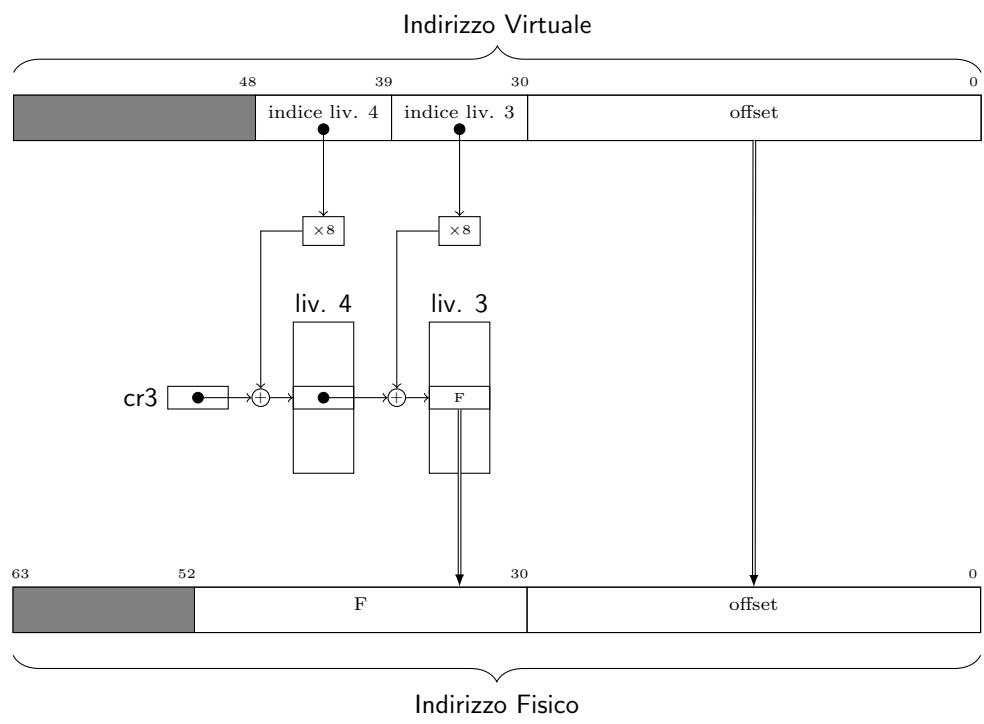


Figura 4: Traduzione da indirizzo virtuale a fisico (pagine di 1 GiB).

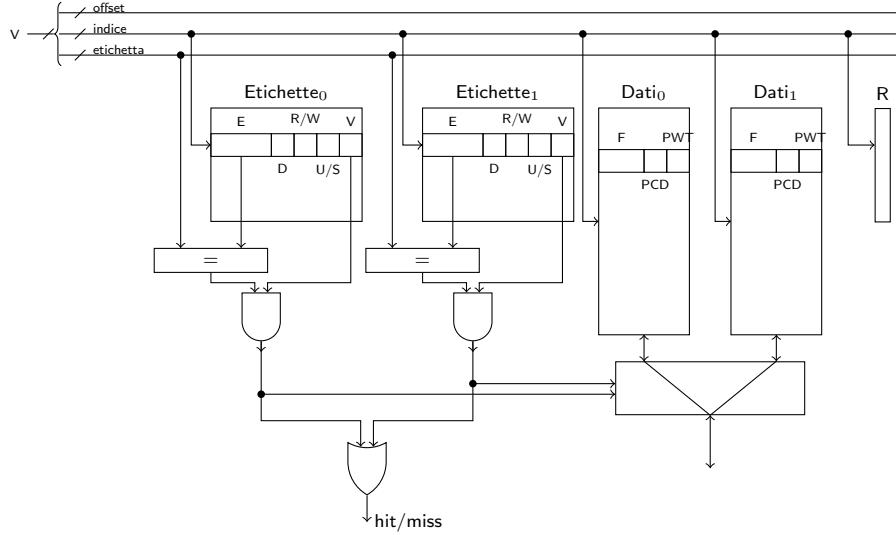


Figura 5: Un esempio di TLB a 2 vie.

cache è di ricordare le *traduzioni* utilizzate più recentemente, dove per traduzioni intendiamo, per le pagine di 4 KiB, quelle definite dai descrittori di livello 1, per le pagine di 2 MiB quelle nei descrittori di livello 2 e, infine, quelle dei descrittori di livello 3 per le pagine da 1 GiB. In ogni caso, per ognuna di queste traduzioni, non ci interessano i descrittori di livello più alto, il cui unico scopo è di permettere di raggiungere, dato l'indirizzo virtuale da tradurre, il descrittore che si trova in fondo alla catena. Una volta raggiunto questo descrittore la MMU può ricordare (nel TLB) quale sia la traduzione da fare per l'indirizzo virtuale in questione: non ha bisogno di ricordare tutto il percorso.

Consideriamo, per il momento, le sole pagine di 4 KiB. In questo caso il TLB è una cache dei descrittori di livello 1. Quando la MMU deve tradurre un indirizzo controlla prima se il TLB contiene già il descrittore che sta cercando; altrimenti si comporta come abbiamo visto fin'ora, seguendo tutta la catena di tabelle dal livello 4 fino al livello 1; alla fine, oltre ad eseguire la traduzione e completare l'accesso in memoria per conto del processore, memorizzerà nel TLB il descrittore appena usato, possibilmente rimpiazzandone un altro. Tipicamente il TLB è una memoria associativa ad insiemi e il descrittore da rimpiazzare sarà scelto in base ad un algoritmo di pseudo-LRU. In Figura 5 è mostrato un esempio di TLB a 2 vie. I campi dati di ogni via memorizzano i campi F dei descrittori di livello 1. La parte offset dell'indirizzo V in ingresso non è utilizzata per accedere alla memoria cache, ma andrà direttamente a far parte dell'indirizzo fisico. Un TLB moderno potrebbe avere 8 vie e contenere 1024 descrittori in totale (quindi $1024/8 = 128$ indici).

Il TLB, come tutte le cache, è trasparente al software, persino al software di sistema: l'architettura non prevede istruzioni che permettano al software di

esaminarne il contenuto. Le uniche istruzioni che l'architettura mette a disposizione sono quelle che permettono di *invalidarlo* in tutto o in parte. Queste operazioni si rendono necessarie quando il software modifica qualcosa nelle tabelle, rendendo dunque non più valide le informazioni che potrebbero trovarsi nel TLB. In particolare, le istruzioni disponibili per l'invalidazione sono due:

- `movq %rax, %cr3`, che già conosciamo; questa istruzione cambia potenzialmente tutta la tabella di livello 4 usata fino al momento prima, quindi tutte le informazioni contenute nel TLB sono da considerarsi non più valide e l'intero TLB viene svuotato;
- `invlpg operando in memoria`: questa istruzione dice al TLB di invalidare la traduzione relativa all'indirizzo dell'operando passato come argomento.

Ogni volta che si cambia processo verrà eseguita una `movq %rax, %cr3` per caricare il puntatore alla tabella di livello 4 del processo entrante. Questa istruzione avrà l'effetto di svuotare il TLB, che è quello che vogliamo, in quanto le traduzioni in esso presenti facevano riferimento alla memoria virtuale del processo uscente.

Il TLB deve permettere alla MMU di svolgere tutte le sue funzioni senza consultare l'albero di traduzione. Ricordiamo che la MMU, oltre ad eseguire la traduzione, deve anche controllare i permessi sui bit U/S e R/W, aggiornare i bit A e D e passare al controllore cache i bit PWT e PCD. Questi ultimi due bit vengono memorizzati insieme al numero di frame e, in caso di hit, passati alla MMU che li girerà al controllore cache.

Consideriamo i bit U/S: la MMU ne incontra fino a quattro durante la traduzione, ma non c'è bisogno che il TLB li ricordi tutti e quattro, in quanto la regola dice che l'accesso è vietato da livello utente se anche uno solo di essi è zero. È dunque sufficiente che la MMU calcoli l'AND dei bit incontrati nel percorso e carichi soltanto questo nel TLB. Lo stesso discorso vale per i bit R/W.

Un discorso diverso va fatto per i bit A e D, in quanto questi bit sono *scritti* dalla MMU durante la traduzione, e non vogliamo certo che la MMU sia costretta a percorrere l'albero per aggiornare questi bit anche quando ha trovato la traduzione nel TLB (cosa che renderebbe il TLB del tutto inutile). Per il bit A la soluzione adottata è semplice: la MMU li aggiorna solo quando non trova la traduzione nel TLB. Fino a che una traduzione si trova nel TLB, i corrispondenti bit A nel percorso di traduzione non verranno ulteriormente modificati dalla MMU. Questo è un problema solo se il software, per qualche motivo, li azzera, perché in quel caso resterebbero a zero anche in presenza di ulteriori accessi. In questo caso deve essere il software stesso che si deve preoccupare di invalidare, in tutto o in parte, il TLB dopo aver azzerato i bit A, costringendo dunque la MMU a ripercorrere l'albero e ri-aggiornare i bit A al prossimo accesso.

Per il bit D il discorso è più complesso, perché può succedere che vi sia un primo accesso in sola lettura all'interno di una pagina, seguito da un accesso in scrittura. Nel primo accesso la MMU non porta, ovviamente, D a 1, e la

traduzione viene caricata nel TLB. Al secondo accesso la MMU trova la traduzione nel TLB e, se non prendessimo provvedimenti, non accederebbe all'albero e lascerebbe il bit D sempre a zero, pur essendovi stata una scrittura all'interno della pagina. L'informazione offerta dal bit D sarebbe quindi inaffidabile. In questo caso è complicato trovare un rimedio puramente software, in quanto il software non sa cosa contenga il TLB. La soluzione adottata è hardware: il TLB ricorda anche il valore del bit D al momento del caricamento della traduzione e causa una *miss* per gli accessi in scrittura con D uguale a 0. Nell'esempio precedente accade dunque questo: nel primo accesso (sola lettura) il TLB carica la traduzione e il valore corrente di D, che è 0. Nel successivo accesso in scrittura il TLB si accorge che, anche se la traduzione è presente, il bit D è ancora a zero. Genera dunque un miss, forzando la MMU a percorrere l'albero, aggiornando di conseguenza il bit D. Si noti che a quel punto la MMU ricaricherà la traduzione nel TLB, questa volta con il bit D a 1: i successivi accessi in scrittura non causeranno dunque ulteriori miss.

Pensiamo ora alle pagine di grandi dimensioni, per esempio le pagine di 2 MiB. Il TLB per le pagine di 4 KiB non è in grado di memorizzare direttamente la traduzione di una pagina più grande, in quanto il numero di bit di offset è diverso. Se vogliamo continuare a usare un unico TLB, la traduzione di una pagina di 2 MiB deve essere riscomposta nelle equivalenti 512 traduzioni di pagine da 4 KiB, occupando dunque 512 posizioni del TLB. Anche se alcuni processori più vecchi adottano questa soluzione, i processori moderni hanno invece un diverso TLB per ogni dimensione di pagina supportata. Si noti che la MMU deve cercare la traduzione in *ciascuno* di questi TLB, perché ogni indirizzo potrebbe essere tradotto usando pagine di qualunque dimensione, e non è possibile saperlo in anticipo. Per fortuna, la ricerca può essere svolta in parallelo, e dunque non influisce negativamente sulle prestazioni. La presenza di questi ulteriori TLB è un motivo in più per usare pagine di grandi dimensioni, quando possibile, in modo da alleggerire il TLB principale.

Funzioni di supporto per la paginazione

G. Lettieri

19 Maggio 2021

La libreria `libcbe` definisce i tipi numerici `paddr` `vaddr`, che rappresentano rispettivamente indirizzi fisici e virtuali. I due tipi hanno solo lo scopo di ricordare al programmatore quando si suppone che un indirizzo debba essere virtuale o fisico, ma sono entrambi equivalenti ad un intero senza segno su 64 bit.

La libreria contiene anche alcune strutture dati e funzioni di uso generale, a cui si può accedere includendo il file `vm.h`.

1 La funzione `norm()`

La funzione `vaddr norm(vaddr a)` serve a *normalizzare* un indirizzo virtuale, cioè a rendere i 16 bit più significativi tutti uguali al bit numero 47. Può essere anche usata per controllare se un dato indirizzo (per esempio ricevuto da una sorgente non fidata, come il livello utente) è normalizzato o meno:

```
if (norm(v) == v) {
    // v e' normalizzato
} else {
    // v non e' normalizzato: errore
}
```

2 La funzione `dim_region()`

La funzione `natq dim_region(int liv)` restituisce la dimensione in byte di una regione di livello `liv`. Si ricordi che abbiamo chiamato “regione di livello i ” l’intervallo di indirizzi coperti da una singola entrata di un tabella di livello $i+1$. Quindi, per esempio, una regione di livello 0 è grande 4096 byte (pagina di livello 1), mentre una regione di livello 1 è grande 2 MiB (pagina di livello 2). La funzione può essere anche utilizzata per ottenere le maschere che permettono di estrarre da un indirizzo virtuale il numero di pagina e l’offset. Per esempio, se `v` è un indirizzo virtuale e vogliamo sapere in che pagina di livello 2 cade, e a quale offset:

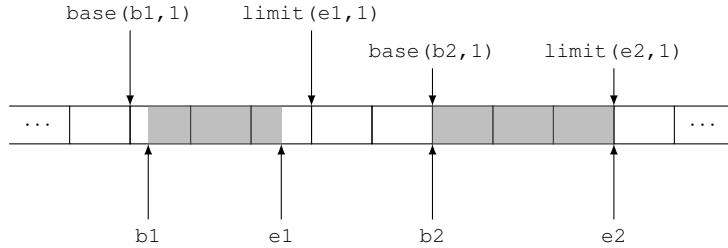


Figura 1: Esempio di calcolo di `base()` e `limit()` per due intervalli di indirizzi, $[b_1, e_1]$ e $[b_2, e_2]$.

```

natq mask = dim_region(liv) - 1;
natq base = v & ~mask; // indirizzo base della pagina
natq offset = v & mask; // offset nella pagina

```

Questo perché `dim_region(liv)` ha la forma di 2^n , che in binario è un 1 seguito da n zeri, e dunque `dim_region(liv) - 1` produce una maschera di n bit ad 1.

3 Le funzioni `base()` e `limit()`

La base della pagina di livello `liv` in cui cade un indirizzo `v` si può ottenere anche dalla funzione `vaddr base(vaddr v, int liv)`. Invece, la funzione `vaddr limit(vaddr e, int liv)` serve a calcolare la base della prima pagina che si trova a destra di un intervallo $[b, e)$ senza toccarlo (si veda la Figura 1).

4 Il tipo `tab_entry`

Il tipo `tab_entry` rappresenta una entrata di una tabella, di qualunque livello. Il file contiene la definizione di un po' di costanti, una per ogni bit del byte di accesso delle entrate, che possono essere usati come maschere per estrarre, settare o resettare i vari bit. Per esempio, se `e` è un riferimento ad una entrata di una tabella, possiamo

- esaminare il bit P con “`if (e & BIT_P) { /*qualcosa */};`”;
- settare il bit U/S con “`e |= BIT_US`”;
- resettare il bit R/W con “`e &= ~BIT_RW`”

e così via.

La funzione `paddr extr_IND_FISICO(tab_entry e)` può essere usate per estrarre l'indirizzo fisico contenuto nell'entrata `e`. Tale indirizzo rappresenta

l'indirizzo (fisico) della tabella di livello inferiore o, nel caso di tabelle foglia, la base del frame in cui è mappata una pagina virtuale.

La funzione `set_IND_FISICO(tab_entry& e, paddr p)` serve invece a settare il campo “numero di frame” dell’entrata `e` con il numero di frame dell’indirizzo fisico `p`, senza modificare il byte di accesso.

5 Le funzioni `i_tab()` e `get_entry()`

La funzione `int i_tab(vaddr v, int liv)` estrae dall’indirizzo virtuale `v` l’indice che la MMU usa per consultare le tabelle di livello `liv`. Per esempio, se `liv` è 2, la funzione restituisce l’indice contenuto nei bit 29–21 di `v`.

La funzione `tab_entry& get_entry(paddr t, int i)`, dato l’indirizzo fisico `t` di una tabella e un indice `i`, restituisce un riferimento all’entrata `i`-esima della tabella stessa (il riferimento potrà essere effettivamente usato solo se è accessibile la finestra sulla memoria fisica).

Per esempio, se vogliamo settare il bit R/W nell’entrata relativa ad un indirizzo virtuale `v` in una tabella di livello 2 di indirizzo fisico `tab`, possiamo scrivere

```
tab_entry& e = get_entry(tab, i_tab(v, 2));
e |= BIT_RW;
```

Come ulteriore esempio, supponiamo che `tab4`, `tab3`, `tab2` e `tab1` siano gli indirizzi fisici di quattro tabelle inizialmente vuote e che vogliamo costruire il percorso di traduzione che mappi l’indirizzo virtuale `v` nell’indirizzo fisico `p`, in modo che sia consentito l’accesso in scrittura ma non quello da livello utente. Possiamo scrivere:

```
// aggancio tab4->tab3
tab_entry& e4 = get_entry(tab4, i_tab(v, 4));
set_IND_FISICO(e4, tab3);
e4 |= BIT_P | BIT_RW;
// aggancio tab3->tab2
tab_entry& e3 = get_entry(tab3, i_tab(v, 3));
set_IND_FISICO(e3, tab2);
e3 |= BIT_P | BIT_RW;
// aggancio tab2->tab1
tab_entry& e2 = get_entry(tab2, i_tab(v, 2));
set_IND_FISICO(e2, tab1);
e2 |= BIT_P | BIT_RW;
// installo la traduzione v -> p
tab_entry& e1 = get_entry(tab1, i_tab(v, 1));
set_IND_FISICO(e1, p);
e1 |= BIT_P | BIT_RW;
```

6 Le funzioni `readCR3()` e `loadCR3()`

Queste funzioni sono scritte in assembler e permettono di leggere e scrivere nel registro **cr3**. In particolare, `loadCR3()` va usata per attivare un nuovo albero di traduzione, passandole l'indirizzo fisico della tabella radice. Si ricordi che ha l'effetto collaterale di invalidare tutto il TLB.

7 Le funzioni per invalidare il TLB

La funzione `invalida_entrata_TLB(vaddr v)` serve a invalidare la traduzione associata all'indirizzo virtuale *v*, nel caso il TLB ne stesse conservando una copia. È scritta in assembler e usa l'istruzione `invlpg`. La funzione va usata ogni qual volta si cambia qualcosa nel percorso di traduzione di *v*. Non solo, dunque, se si cambia la traduzione, ma anche se si cambia qualche bit di uno qualunque dei byte di accesso che si incontrano nel percorso di traduzione di *v*, perché il TLB memorizza anche quelli, o comunque assume che siano sempre nello stato in cui li aveva visti la MMU al momento del caricamento della traduzione.

La funzione `invalida_TLB()` serve ad invalidare tutto il contenuto del TLB. È equivalente a `loadCR3(read(CR3))`. Ha senso chiamarla se sono stati fatti molti cambiamenti (per esempio, azzeramento di tutti i bit A dopo averli esaminati) e dunque diventa conveniente rispetto a chiamare tante volte `invalida_entrata_TLB()`.

8 L'iteratore `tab_iter`

La struttura dati `tab_iter` è probabilmente la funzionalità più utile offerta dalla `libce`. Si tratta di un iteratore che permette di visitare tutte le entrate dell'albero di traduzione coinvolte, a tutti i livelli, nella traduzione di tutti gli indirizzi di un dato intervallo. La visita è del tipo *depth first* e può essere eseguita sia in ordine anticipato che posticipato. Tutte le entrate sono visitate una sola volta e le entrate foglia (che contengono le traduzioni) sono visitate in base all'ordine crescente degli indirizzi virtuali.

L'iteratore va costruito specificando l'indirizzo fisico della tabella radice dell'albero, la base dell'intervallo e la sua lunghezza (1 per default). Ad ogni istante, a meno ché la visita non sia terminata, l'iteratore si trova su una qualche entrata di una qualche tabella dell'albero. Appena costruito si troverà sull'entrata della tabella radice relativa all'indirizzo base dell'intervallo da visitare. L'iteratore può essere spostato sulla prossima entrata (secondo l'ordine *depth first*) con il metodo `next()`. L'operatore di conversione a `bool` restituisce `false` quando la visita è terminata. Le funzioni membro `get_e()`, `get_tab()`, `get_l()` e `get_e()` permettono di ottenere, rispettivamente, un riferimento all'entrata su cui si trova l'iteratore, l'indirizzo fisico della tabella che contiene questa entrata e il livello (4, 3, 2 o 1) di questa tabella. La funzione `get_v()`, invece, restituisce il più piccolo indirizzo virtuale la cui traduzione passa da questa entrata.

Consideriamo prima il caso particolare in cui l'intervallo consiste di un unico indirizzo e rifacciamoci all'esempio alla fine della Sezione 5. Possiamo stampare tutto il percorso di traduzione di v nel seguente modo:

```
for (tab_iter it(tab4, v); it; it.next()) {
    printf("tab %x, liv %d, entry %x\n",
        it.get_tab(),
        it.get_l(),
        it.get_e());
}
```

Il ciclo **for** costruisce un iteratore per l'albero di radice tab4 e per l'intervallo di indirizzi virtuali [v, v + 1) (non avendo passato la lunghezza dell'intervallo come terzo argomento viene assunto il default di 1). Nella prima iterazione it si trova sull'entrata di tab4 che abbiamo chiamato e4 nella sezione 5. La printf() stamperà dunque l'indirizzo tab4, il livello 4 e il contenuto di e4, vale a dire l'indirizzo tab3 e il byte di accesso. Nella seconda iterazione it si sposterà su e3 e la printf() stamerà l'indirizzo tab3, il livello 3 e il contenuto di e3, cioè l'indirizzo tab2 e il byte di accesso. Lo stesso accadrà per il livello 2 e il livello 1. A quel punto la visita è terminata e l'espressione "it" (che invoca l'operatore di conversione a **bool**) restituirà **false**, terminando il ciclo.

Supponiamo di modificare il codice qui sopra cambiando il ciclo for in

```
for (tab_iter it(tab4, v, 2*DIM_PAGINA); it; it.next()) {
```

Ora vogliamo visitare le traduzioni delle due pagine v e v+DIM_PAGINA (la pagina successiva a v, supponendo che v non sia l'ultima pagina dello spazio di indirizzamento e non sia adiacente al "buco" nello spazio). Supponendo che l'albero sia sempre quello creato in Sezione 5, l'iteratore seguirebbe lo stesso percorso di prima e, dopo aver visitato e1, si sposterebbe sull'entrata di tab1 successiva, oppure, se e1 fosse l'ultima entrata di tab1 (quella di indice 511), sull'entrata di tab2 successiva a e2 (o ancora più su, se anche e2 fosse l'ultima entrata di tab2). La printf() mostrerebbe la tabella e il livello su cui l'iteratore si è fermato e il contenuto dell'entrata corrente, che nel nostro caso sarebbe tutto nullo. Al prossimo passo la visita sarebbe terminata, perché la pagina v+DIM_PAGINA non ha una traduzione e non ci sono altre pagine nell'intervallo. Se invece anche v+DIM_PAGINA avesse una traduzione, l'iteratore scenderebbe lungo il suo percorso, fermandosi ad ogni livello fino al livello 1, e solo allora terminerebbe la visita.

La visita in ordine anticipato è utile quando vogliamo *costruire* l'albero di traduzione per un certo intervallo di indirizzi. Ogni volta che l'iteratore si ferma possiamo esaminare il bit P dell'entrata corrente e, se vale 0 e non siamo ancora arrivati al livello 1, allocare e agganciare una nuova tabella di livello inferiore. Per far questo sfruttiamo il fatto che get_e() ci restituisce un *riferimento* all'entrata su cui si trova l'iteratore, permettendoci dunque di modificarla. Al prossimo passo l'iteratore si sposterà sulle entrate rilevanti di questa nuova tabella e noi potremo continuare ad allocare ed agganciare le tabelle di livello

inferiore oppure, arrivati al livello 1, installare le traduzioni. Qui possiamo usare la funzione membro `get_v()` per chiedere all'iteratore qual è l'indirizzo virtuale il cui percorso di traduzione porta all'entrata su cui ci troviamo, in modo da poter scegliere correttamente la traduzione da associarvi. Questo è il meccanismo usato dalla funzione `map()` del modulo sistema.

L'iteratore può essere usato anche per eseguire una visita in ordine posticipato, nel seguente modo

```
tab_iter it(tab4, v);
for (it.post(); it; it.next_post()) {
    printf("tab %x, liv %d, entry %x\n",
        it.get_tab(),
        it.get_l(),
        it.get_e());
}
```

In questo caso il codice mosterà le entrate del percorso partendo dal livello 1 e salendo fino al 4. La visita in ordine posticipato è utile quando vogliamo *distruggere* un albero di traduzione, in quanto ci permette di eliminare i livelli inferiori dell'albero prima di esaminare i livelli superiori. Questo è il meccanismo usato dalla funzione `unmap()` del modulo sistema.

Quando vogliamo esaminare il percorso di traduzione di un singolo indirizzo conviene usare il seguente codice

```
tab_iter it(tab4, v);
while (it.down())
;
```

La funzione membro `down()` prova soltanto a scendere nell'albero, seguendo il percorso di traduzione di `v`, fermandosi alla prima foglia (sia perché ha trovato un bit P a zero, sia perché è arrivata alla traduzione). All'uscita dal **while** l'iteratore si trova ancora sull'entrata foglia, che possiamo così esaminare e/o modificare. Questo non è sempre vero con gli altri tipi di visita.

Implementazione della memoria virtuale

G. Lettieri

19 Maggio 2021

In questa ultima parte studiamo l'implementazione della paginazione nel nucleo didattico.

1 La memoria virtuale nel nucleo

Realizzeremo un caso ibrido, in cui i processi hanno sia zone di memoria condivise tra tutti, sia zone di memoria private per ciascuno. Tutti i processi utente avranno caricato nella propria memoria virtuale un unico programma: quello contenuto nel modulo utente. Ogni processo, però, partì eseguendo una funzione del programma che può anche essere diversa per ciascuno.

La memoria virtuale di ogni processo sarà organizzata come in Figura 1. La Figura mostra lo spazio di indirizzamento virtuale di due processi, P_1 e P_2 , insieme al possibile contenuto della memoria fisica. Le annotazioni che si trovano ai lati sinistro di P_1 valgono anche per P_2 e per qualunque altro processo, e lo stesso per le annotazioni che si trovano a destra di P_2 . La memoria virtuale di ogni processo è suddivisa *a priori* nello stesso modo, come mostrato sulla destra della memoria virtuale di P_2 : la parte che va dall'indirizzo 0000 0000 0000 0000 all'indirizzo 0000 7fff ffff ffff ($2^{47} - 1$) è dedicata al sistema (bit U/S pari a 0 in tutte le traduzioni), mentre la parte che va da ffff 8000 0000 0000 a ffff ffff ffff ffff è dedicata all'utente (bit U/S pari a 1 in tutte le traduzioni). Gli indirizzi virtuali della prima pagina (da 0 a fff) sono lasciati non mappati, per intercettare dereferenziazioni di nullptr indipendentemente dal livello di privilegio del processore.

Ogni parte (utente o sistema) dello spazio di indirizzamento di ogni processo è suddivisa in ulteriori sezioni. Sulla sinistra della memoria virtuale di P_1 abbiamo mostrato alcune costanti (definite in `sistema.cpp`) che contengono gli indirizzi di inizio e fine delle varie sezioni. Per “indirizzo di fine” intendiamo il primo indirizzo che non fa parte della sezione: gli indirizzi di una sezione vanno da quello di inizio, incluso, a quello di fine, escluso. Il nome delle costanti è composto da tre parti: 1) la stringa “ini” per l'indirizzo di inizio o “fin” per l'indirizzo di fine; 2) la stringa “sis” per le sezioni *sistema*, “mio” per la sezione *modulo I/O* e la stringa “utn” per le sezioni *utente*; 3) il carattere “c” per le sezioni *condivise* o “p” per quelle private. Quindi, per

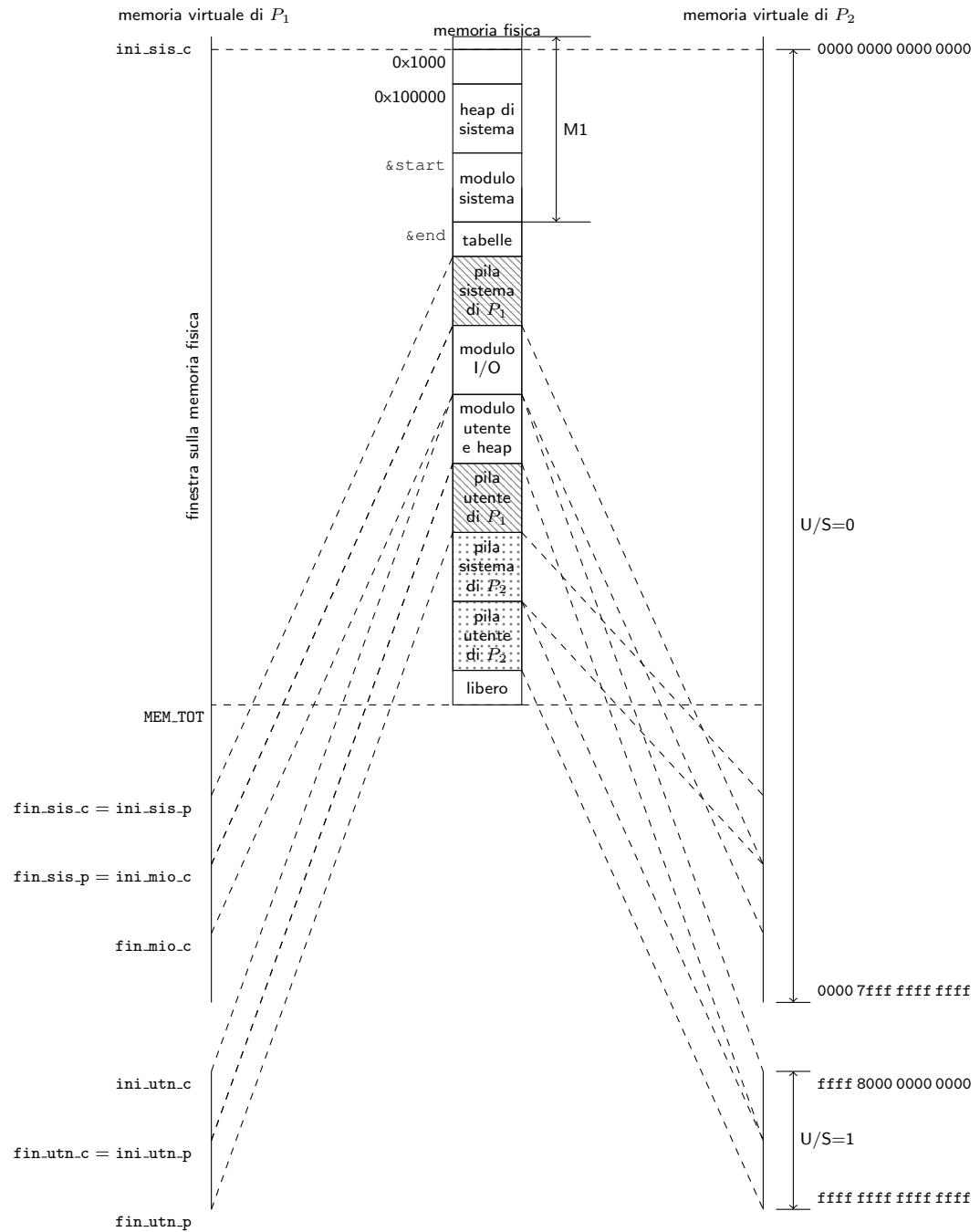


Figura 1: Esempio di memoria virtuale con due processi (figura non in scala).

esempio, `fin_sis_c` è l'indirizzo di fine della sezione sistema/condivisa. Le sezioni condivise contengono le stesse traduzioni in tutti i processi.

Le sezioni della memoria virtuale di ogni processo sono elencate di seguito.

sistema/condivisa: contiene la finestra sulla memoria fisica. La dimensione della sezione è decisa *a priori*, mentre la dimensione della finestra dipende dalla memoria fisica installata e può essere più piccola (come illustrato in Figura, la finestra arriva solo fino a `MEM_TOT`). La sezione contiene anche delle traduzioni identità (non mostrate in Figura) per quegli indirizzi che corrispondono a registri di periferiche (come per esempio l'APIC). Il resto della sezione è lasciato inutilizzato.

sistema/privata: contiene la pila sistema del processo. Si noti che questa pila è utilizzata sia dalle funzioni del modulo sistema, sia da quelle del modulo I/O, in quanto questo gira a livello sistema e ogni processo ha una sola pila di livello sistema.

IO/condivisa: contiene il modulo I/O, vale a dire le sezioni `.text`, `.data` estratte dal file `build/io` e mappate al loro indirizzo di collegamento. Contiene inoltre lo *heap I/O*, vale a dire la zona di memoria su cui lavorano gli operatori `new` e `delete` quando usati dal modulo I/O.

utente/condivisa: contiene il modulo utente, vale a dire le sezioni `.text` e `.data` estratte dal file `build/utente` e caricate al loro indirizzo di collegamento. Analogamente al modulo I/O, contiene inoltre lo *heap utente*, che è usato dagli operatori `new` e `delete` quando sono invocati dal modulo utente.

utente/privata: contiene la pila utente del processo.

La memoria fisica è suddivisa, come al solito, in una parte M1 e una parte M2 (per motivi di spazio, solo la parte M1 è indicata esplicitamente in Figura; la parte M2 è quella rimanente). La suddivisione della parte M2 della memoria fisica in Figura 1 è puramente esemplificativa: i processi saranno in generale più di due; inoltre, se osserviamo il sistema ad un qualunque istante, potremo trovare le varie parti in un ordine qualsiasi e in frame non contigui della memoria fisica.

La parte M1 della memoria fisica è organizzata nel modo seguente.

- Il primo MiB è riservato per ragioni storiche (parte degli indirizzi sono occupati da altre cose, come la memoria video in modalità testo, che si trova all'indirizzo `b8000`).
- La parte che va dalla fine del primo MiB all'inizio del modulo sistema contiene lo *heap di sistema*. Si tratta di una zona di memoria in cui le primitive del modulo sistema allocano i `des_proc`, le strutture `richiesta` (usate dalla primitiva `delay`), e in generale qualunque struttura dati che debba essere allocata dinamicamente (tramite gli operatori `new` e `delete`).

- La parte dedicata al modulo sistema contiene più precisamente le sezioni `.text` e `.data` estratte dal file `build/sistema` e caricate al loro indirizzo di collegamento. La fine della parte precedente e l'inizio della successiva verranno ottenuti, a tempo di inizializzazione, dai due simboli `start` ed `end` definiti nel modulo sistema.

Le linee tratteggiate tra le memorie virtuali di P_1 e P_2 e la memoria fisica vogliono rappresentare la corrispondenza tra le sezioni della memoria virtuale di ciascun processo e le parti di memoria fisica in cui sono tradotte. Le parti a sfondo bianco delle memoria fisica corrispondono a sezioni condivise tra tutti i processi. Si noti, per esempio, come le sezioni “modulo I/O” e “modulo utente e heap” corrispondano alla stessa parte della memoria fisica sia per P_1 che per P_2 . Le sezioni private, invece, usano traduzioni diverse in ciascun processo e corrispondono a parti diverse della memoria fisica: si veda per esempio la sezione utente/privata (tra gli indirizzi `ini_utn_p` e `fin_utn_p`) che corrisponde a “pila utente di P_1 ” per il processo P_1 e “pila utente di P_2 ” per il processo P_2 ; un discorso analogo vale per la parte sistema/privata. Si noti che alcune parti della memoria fisica sono accessibili esclusivamente tramite la finestra: in particolare, tutta la parte M1 e i frame di M2 che o contengono tabelle o sono vuote (la parte indicata con “libero” in Figura 1). Alcune parti di M2 sono accessibili sia dalla finestra, sia da altre sezioni. Ciò è dovuto semplicemente al fatto che la finestra permette di accedere a tutta la memoria fisica, indipendentemente dal contenuto, e dunque contiene traduzioni anche per tutti i frame che contengono le pagine della memoria virtuale di ogni processo.

2 Implementazione

In questa sezione esaminiamo le strutture dati e le funzioni che si trovano nel modulo sistema e sono relative all’implementazione della memoria virtuale.

La suddivisione dello spazio di indirizzamento dei processi nelle sue varie parti è stabilita da alcune costanti definite in `costanti.h` (si vedano i commenti in “suddivisione della memoria virtuale” all’interno del file). Per semplicità le varie parti occupano ciascuna un numero intero di regioni di livello massimo (quindi 512 GiB per i trie a 4 livelli), e dunque le possiamo distinguere in base alle entrate utilizzate nella tabella radice. Per esempio, la parte sistema/condivisa usa la prima entrata della radice (indice 0), mentre la parte sistema/privata usata la seconda (indice 1).

Anche le dimensioni delle varie parti *fisiche* della memoria dei processi sono decise *a priori* con delle costanti: la dimensione delle pila, sistema e utente, e la dimensione degli heap dei tre moduli (sistema, I/O e utente). Lo stesso vale per la RAM totale del sistema, che è decisa dalla costante `MEM_TOT`. Per aumentare o diminuire la RAM usata dal sistema ($M1+M2$) occorre modificare questa costante e ricompilare. Si noti che lo script `run` configura la macchina virtuale con la quantità di RAM stabilita da questa stessa costante.

```

1  struct des_frame {
2    union {
3      natw nvalide;
4      natl prossimo_libero;
5    };
6  };

```

Figura 2: Il descrittore di frame.

2.1 Gestione della memoria fisica

Il sistema deve sapere quali frame della memoria sono liberi o occupati, in modo da poterli allocare per le pagine e le tabelle dei processi. Dal momento che tutti i frame sono equivalenti, è sufficiente mantenere una lista dei frame liberi da cui si estrae e si inserisce sempre in testa.

In generale, il sistema ha bisogno di associare anche altre informazioni ad ogni frame. Per questo scopo il modulo sistema alloca un *descrittore di frame* per ogni frame della memoria fisica. Il descrittore ha il tipo mostrato in Figura 2 e contiene le seguenti informazioni:

- `prossimo_libero`: (significativo solo se il frame è libero) il numero del prossimo frame nella lista dei frame liberi;
- `nvalide`: (significativo solo se il frame contiene una tabella) quante entrate con P diverso da zero sono contenute nella tabella.

Si noti che, dal momento che le due informazioni non sono mai utili contemporaneamente, sono dichiarate all'interno di una `union`, che permette di risparmiare spazio.

I descrittori di frame sono raccolti in un array, `vdf[]`, indicizzato dal numero di frame. La funzione `init_frame()`, chiamata in fase di inizializzazione, si preoccupa di inizializzare questo array, inserendo tutti i frame di M2 nella lista dei frame liberi. Da questo punto in poi è possibile allocare e deallocare frame usando le funzioni:

- `paddr alloca_frame()`, che prova ad estrarre un frame dalla lista dei liberi e ne restituisce l'indirizzo (fisico) di base (0 se la lista è vuota);
- `void dealloca_frame(paddr p)`, che inserisce il frame di indirizzo fisico `p` nella lista dei liberi.

Le funzioni di utilità `inc_ref()`, `dec_ref()` e `get_ref()` servono a manipolare il contatore `nvalide` di una tabella di cui conosciamo l'indirizzo fisico.

2.2 Creazione delle parti condivise

Tutte le traduzioni relative alle parti condivise dei processi (sistema, I/O e utente) possono essere create una volta per tutte all'avvio del sistema. I processi

possono condividere tutto il sottoalbero che implementa queste traduzioni, a partire dal livello inferiore a quello della radice (il livello 3 nei processori con trie a 4 livelli). Per esempio, possiamo creare la traduzione della finestra sulla memoria una sola volta. Tutte le entrate di indice 0 delle tabelle radice di tutti i processi punteranno poi alla stessa tabella di livello inferiore, e dunque tutto il sottoalbero sarà condiviso tra tutti i processi. Questo ci permette di risparmiare molto spazio, e anche del tempo ogni volta che creiamo un nuovo processo.

Le traduzioni della parte sistema/condivisa sono create dal modulo sistema nella fase di inizializzazione, tramite la funzione `crea_finestra_FM()`. In particolare sono mappati in se stessi tutti gli indirizzi da `DIM_PAGINA` a `MEM_TOT`, in modo da creare la finestra sulla memoria fisica, lasciando non mappata la prima pagina in modo da poter intercettare le dereferenziazioni di `nullptr`. Le traduzioni della finestra sono implementate con pagine di livello 2 dove possibile, e sono impostate con bit R/W a 1 (scrittura permessa) e bit U/S a zero (accesso consentito solo da livello sistema). Questa traduzione permette anche di accedere alle sezioni `.text`, `.data` e `.bss` del modulo sistema, che sono collegate e caricate a partire dall'indirizzo 200000 (2 MiB, inizio del terzo megabyte di RAM). Si noti che la finestra comprende anche la memoria video in modalità testo, che si trova nella pagina di indirizzo base `b8000`. Nella traduzione degli indirizzi di questa pagina viene anche settato il bit PWT, in modo che le scritture raggiungano effettivamente la memoria video e non si fermino in cache per un tempo indefinito. Non è invece necessario disabilitare del tutto la cache, in quanto nel nostro caso la memoria video può essere modificata solo dal software e, dunque, possiamo sfruttare la cache per le operazioni di lettura. Sempre nella stessa funzione sono anche create le traduzioni identità che permettono di accedere all'APIC, i cui registri, come sappiamo, sono mappati nello spazio di memoria. In questo caso dobbiamo completamente disabilitare la cache, settando il bit PCD.

Le traduzioni delle parti IO/condivisa e utente/condivisa sono create nella funzione `crea_spazio_condiviso()`. Il boot loader di QEMU ha copiato i file `io` e `utente` in memoria (nel secondo megabyte di RAM). Gli indirizzi di partenza delle due copie sono scritti in una struttura dati che il boot loader passa alla routine di inizializzazione del modulo sistema¹. La vera e propria creazione delle traduzioni per i due moduli si trova nella funzione `carica_modulo()`, invocata una volta per il modulo I/O e un'altra per il modulo utente. Questa funzione interpreta il file ELF copiato in memoria e ne consulta la tabella di programma, per sapere quali parti del file devono essere mappate nello spazio di indirizzamento. Per ciascuna di esse alloca opportunamente dei frame dalla parte M2, vi copia il corrispondente contenuto del file (o provvede ad azzerare i byte del frame, nel caso del `.bss`) e crea le traduzioni. Inoltre, alloca ulteriori frame destinati a contenere lo heap del modulo e li mappa in coda all'ultimo indirizzo virtuale menzionato nel file ELF (normalmente, dunque, subito dopo

¹In realtà c'è un passaggio intermedio tramite il boot loader della `libce`, in quanto il boot loader di QEMU porta il processore soltanto nella modalità a 32 bit. Il boot loader di `libce` passa alla modalità a 64 bit e cede il controllo al modulo sistema, passandogli gli stessi parametri ricevuti dal boot loader di QEMU.

le sezioni **.data** e **.bss**). A questo punto lo spazio occupato dai due file copiati dal boot loader di QEMU può essere riutilizzato ed entra a far parte dello heap del modulo sistema, che si estende per tutto il secondo megabyte.

Tutte queste traduzioni vengono create nello stesso albero di traduzione. Una volta create, la radice dell'albero viene caricata nel registro **cr3**. Da quel punto in poi il sistema smette di usare le traduzioni che aveva preparato il boot loader e usa soltanto le proprie.

2.3 Creazione dei processi

La creazione dei processi si trova nella funzione `crea_processo()`. Oltre ad allocare e inizializzare un nuovo `des_proc` nei modi che già conosciamo, la funzione si deve preoccupare di creare tutta la memoria virtuale del processo. Per far questo alloca una nuova tabella radice e vi copia tutte le entrate relative alle parti condivise, prendendole dalla tabella radice che in questo momento è puntata da **cr3**. Nel caso dei primi processi, creati in fase di inizializzazione, questa è la tabella radice di cui abbiamo parlato nella sezione precedente. Tutti gli altri processi sono creati da altri processi, e in questo caso il registro **cr3** punta alla tabella radice del processo genitore. In ogni caso, l'effetto è che le entrate relative alle parti condivise punteranno tutte agli stessi sottoalberi, come avevamo anticipato.

La funzione `crea_processo()` deve poi creare le parti private. Queste comprendono la pila sistema (parte sistema/privata) e, per i processi di livello utente, anche la pila utente (parte utente/privata). Queste sono create allocando un numero prestabilito di frame (in base alle costanti definite in `costanti.h`) e poi mappandoli contiguamente partendo dagli indirizzi inferiori delle rispettive parti dello spazio di indirizzamento.

La funzione, ricordiamo, deve anche *inizializzare* la pila sistema, inserendovi le cinque parole quadruple che verranno poi lette dalla **iretq** la prima volta che il processo andrà in esecuzione. Qui bisogna fare particolare attenzione: la funzione non può scrivere nella pila sistema del processo appena creato usando gli indirizzi della parte sistema/privata, in quanto al momento è ancora attivo lo spazio di indirizzamento del processo *genitore* e quegli indirizzi verrebbero tradotti dalla MMU nei frame della pila sistema di questo processo, e non del processo creato. Per poter scrivere nella pila sistema del processo creato, la funzione sfrutta la finestra sulla memoria fisica, accedendo direttamente ai frame della pila tramite il loro indirizzo fisico.

2.4 Le funzioni di supporto

Il modulo sistema definisce alcune funzioni di supporto per svolgere i suoi compiti. Le illustriamo in quanto possono essere utili anche nello svolgimento degli esercizi.

Come abbiamo visto, il sistema associa ad ogni tabella un contatore delle entrate valide. Per gestire correttamente questo contatore, sono utili le seguenti funzioni:

- **paddr alloca_tab()**
alloca un frame destinato a contenere una tabella; rispetto ad una semplice `alloca_frame()` provvede anche ad azzerare sia il frame, sia il contatore `nvalide` nel suo descrittore;
- **void rilascia_tab()**
rilascia un frame che conteneva una tabella; rispetto ad una semplice `rilascia_frame()`, controlla anche che il contatore `nvalide` non sia diverso da zero, in modo da intercettare errori di programmazione in cui si tenta di rilasciare tabelle che ancora contengono entrate valide;
- **void set_entry(paddr tab, natl j, tab_entry se)**
fa il contrario della `get_entry()` di libce, settando l'entrata `j`-esima della tabella di indirizzo fisico `tab` con il nuovo valore `se`; si preoccupa di aggiornare opportunamente il contatore `nvalide` se il valore del bit P cambia per effetto dell'assegnamento;
- **void copy_des(paddr src, paddr dst, natl i, natl n)**
copia `n` entrate a partire dalla `i`-esima della tabella di indirizzo fisico `src` nelle corrispondenti entrate della tabella di indirizzo fisico `dst`; si preoccupa di aggiornare opportunamente il contatore `nvalide` della tabella `dst`; questa funzione è utile, per esempio, quando si crea un nuovo processo e si devono copiare le entrate delle parti condivise dalla tabella radice del processo genitore;
- **void set_des(paddr dst, natl i, natl n, tab_entry e)**
setta `n` entrate a partire dalla `i`-esima della tabella di indirizzo fisico `dst`, impostandole tutte uguali a `e` e aggiornando opportunamente il contatore `nvalide`; questa funzione è utile soprattutto quando `e` è zero perché si sta distruggendo un albero di traduzione (per esempio, quando un processo termina);

Le funzioni di supporto più utili sono le funzioni `map()` e `unmap()`, che il sistema usa per creare (o distruggere) le traduzioni di tutte le parti sistema, IO e utente, sia condivise che private.

La funzione `map()` riceve l'indirizzo fisico `tab` della tabella radice di un trie, gli estremi di un intervallo `[begin, end]` di indirizzi virtuali (allineati alla pagina) e un parametro template `getpaddr` che si deve comportare come una funzione da `vaddr` a `paddr`. La funzione `map()` creerà, nell'albero di radice `tab`, le traduzioni $v \mapsto \text{getpaddr}(v)$ per tutti gli indirizzi di pagina `v` nell'intervallo `[begin, end]`. La funzione riceve anche un parametro `flags` con cui si può specificare il valore desiderato per i flag U/S, R/W, PWT e PCD. Per esempio, per creare delle traduzioni identità nell'intervallo [1000, 80 0000] (esadecimale) in modo che siano accessibili in scrittura da livello sistema, si può scrivere:

```
paddr identity_map(vaddr v)
{
    return v;
```

```

}

void some_function()
{
    ...
    map(tab, 0x1000, 0x800000, BIT_RW, identity_map);
    ...
}

```

La funzione creerà il mapping

$$1000 \mapsto \text{identity_map}(1000) = 1000,$$

poi il mapping

$$2000 \mapsto \text{identity_map}(2000) = 2000$$

e così via fino a

$$7f\ 0000 \mapsto \text{identity_map}(7f\ 0000) = 7f\ 0000.$$

Se invece vogliamo mappare gli stessi indirizzi su dei nuovi frame di M2, basta sostituire la funzione passata come `getpaddr()`:

```

paddr my_alloc_frame(vaddr v)
{
    return alloca_frame();
}

void some_function()
{
    ...
    map(tab, 0x1000, 0x800000, BIT_RW, my_alloc_frame);
    ...
}

```

In questo caso `map()` allocherà un nuovo frame per ogni indirizzo di pagina nell'intervallo, quindi mapperà la pagina su quel frame.

Nel modulo sistema si preferisce usare espressioni *lambda* al posto dei puntatori a funzione, per comodità. Un'altra possibilità è di usare oggetti istanza di classi/strutture che ridefiniscono `operator()`. Questo è utile quando, per creare correttamente le traduzioni, non ci basta conoscere l'indirizzo virtuale e abbiamo bisogno di portarci dietro altre informazioni. Un esempio, nel modulo sistema, è dato dalla funzione `carica_modulo()`, che deve creare un mapping per ogni segmento di un file ELF. Qui vediamo un esempio più semplice: supponiamo di dover creare un mapping tra lo stesso intervallo di prima e degli indirizzi fisici arbitrari, scritti in un array `paddr a[]`. Possiamo operare così:

```

class my_addrs {
    paddr *pa;
}

```

```

int i;
public:
    my_addrs (paddr *pa_) : pa(pa_), i(0) {}

    paddr operator() (vaddr v) {
        return pa[i++];
    }
};

void some_function()
{
    paddr a[] = { ... };
    my_addrs m(a);

    map(tab, 0x1000, 0x800000, BIT_RW, m);
}

```

Opzionalmente, `map()` può creare le traduzioni usando pagine di livello maggiore di 1. È sufficiente passare il livello desiderato come ulteriore parametro. Per esempio, la funzione `crea_finestra_FM()` usa pagine di livello 2 e passa questo valore come ultimo argomento di `map()`.

La funzione `unmap()` esegue l'operazione inversa di `map()`: distrugge tutti le traduzioni in un dato intervallo di indirizzi virtuali. La funzione si preoccupa di deallocare (tramite `rilascia_tab()`) anche tutte le tabelle che diventano vuote dopo aver eliminato le traduzioni dell'intervallo. La funzione riceve un parametro template `putpaddr` che l'utente può usare per decidere cosa fare di ogni indirizzo fisico che prima era mappato da qualche indirizzo virtuale nell'intervallo. Per esempio, per distruggere il mapping creato tramite `identity_map()` non è necessario fare niente e `putpaddr` può essere una NOP:

```

void do_nothing(paddr p, int lvl)
{
}

void some_function()
{
    ...
    unmap(tab, 0x1000, 0x800000, do_nothing);
    ...
}

```

Invece, per disfare i mapping creati tramite `my_alloc_frame()` è necessario chiamare `rilascia_frame()` su tutti i frame non più mappati:

```

void my_rel_frame(paddr p, int lvl)
{
    rilascia_frame(p);
}

```

```
}

void some_function()
{
    ...
    unmap(tab, 0x1000, 0x800000, my_rel_frame);
    ...
}
```

Si noti che la funzione passata a `unmap()` riceve anche un parametro `int lvl`, che è il livello della pagina che era precedentemente mappata sull'indirizzo fisico `p`, nel caso questa informazione sia necessaria per capire cosa fare di `p`.

I/O nel nucleo

G. Lettieri

10 Maggio 2022

Come abbiamo visto nell'esempio che motivava l'introduzione della protezione, le operazioni di I/O offrono un'ottima opportunità per sfruttare appieno l'ambiente multiprogrammato. Ricordiamo brevemente gli aspetti essenziali dell'esempio. Tipicamente, un processo che inizia una operazione di I/O non può proseguire finché l'operazione non è terminata, e d'altro canto l'operazione stessa è normalmente lenta e non ha bisogno della CPU per essere portata avanti, o ne ha bisogno in minima parte. L'idea è quindi di *bloccare* i processi che iniziano una operazione di I/O e sbloccarli (riportarli in coda pronti) quando l'operazione è terminata. In questo modo altri processi potranno andare in esecuzione mentre è in corso l'operazione di I/O e la CPU sarà sfruttata più efficientemente. Si noti che il sistema deve sapere che l'operazione è completata mentre è in esecuzione un processo che, in generale, è completamente scorrelato da essa. Il completamento dell'operazione dovrà essere segnalato da una interruzione, in modo che il sistema possa riacquistare il controllo della CPU e svolgere le operazioni necessarie, tra cui sbloccare il processo che aveva richiesto l'operazione.

Quanto appena descritto è un esempio di *sincronizzazione*: vogliamo che il processo che ha iniziato l'I/O possa proseguire solo dopo che l'operazione di I/O è stata completata.

In ambiente multiprocesso il sistema deve anche preoccuparsi di coordinare l'accesso alle periferiche. Tipicamente, mentre una periferica è impegnata in una operazione di I/O, non può essere usata per altre operazioni. Se un processo vuole usare una periferica mentre questa è occupata, deve aspettare che la precedente operazione termini e la periferica ritorni libera. A differenza del precedente, questo non è un problema di sincronizzazione, ma di *mutua esclusione*: se due processi, P_1 e P_2 , vogliono entrambi usare la stessa periferica, è per noi indifferente se la usa prima P_1 e poi P_2 o viceversa: l'unica cosa che ci interessa è che non la usino contemporaneamente. Si noti infine che la mutua esclusione possiamo garantirla separatamente per ogni periferica: se P_1 e P_2 devono usare due periferiche distinte non hanno bisogno di coordinarsi.

Ricordiamo ora che i processi di cui stiamo parlando sono processi *utente* e, come al solito, sono da considerarsi non fidati. Non possiamo dunque aspettarci che gli utenti si coordinino tra di loro per usare le periferiche uno alla volta, o che sospendano volontariamente i propri processi per far andare avanti quelli

degli altri. Per imporre la mutua esclusione e la sincronizzazione di cui sopra procediamo come al solito:

- impediamo agli utenti di parlare direttamente con le periferiche e
- forniamo delle primitive per svolgere le operazioni di I/O sotto il controllo del sistema.

Per realizzare il primo punto facciamo in modo che il campo IOPL del registro **rflags** dei processi utente specifichi il valore “sistema”. In questo modo, mentre il processore si trova a livello utente, genera una eccezione ogni volta che si prova ad eseguire una istruzione di `in` o `out`¹. Notare che, indipendentemente da questo problema, siamo sostanzialmente obbligati a settare il campo IOPL a “sistema”, in quanto questo è anche il modo in cui vietiamo agli utenti l’utilizzo delle istruzioni `sti` e `cli`, che abilitano e disabilitano le interruzioni esterne mascherabili. Per vietare l’accesso alle periferiche che hanno i registri mappati in memoria ricorriamo invece alla MMU, in uno dei due modi possibili: o non inserendo traduzioni che portino ai registri delle periferiche, o proteggendo le traduzioni con i bit “S/U” nei descrittori.

Occupiamoci ora della struttura generale delle primitive di I/O che il sistema deve fornire. Una tipica primitiva di lettura avrà una interfaccia simile a questa:

```
extern "C" void read_n(nat1 id, char* buf,  
                      natq quanti);
```

La primitiva riceve un parametro `id`, che serve ad identificare la periferica da cui il processo vuole leggere, e un indirizzo `buf` che punta ad un buffer in cui l’utente vuole ricevere i dati. Faremo l’ipotesi che i dati siano sempre una sequenza di byte. Il parametro `quanti` specifica il numero di byte che l’utente vuole leggere. È l’utente che deve preoccuparsi di dichiarare un buffer grande a sufficienza per contenere i byte richiesti. Vedremo che il sistema può imporre altre limitazioni su questo buffer.

Analogamente, la tipica primitiva di scrittura avrà questa interfaccia verso gli utenti:

```
extern "C" void write_n(nat1 id, const char* buf,  
                        natq quanti);
```

I parametri hanno lo stesso significato del caso precedente, ma ora il buffer puntato da `buf` deve contenere i dati che l’utente vuole scrivere (la primitiva si limita a leggerli, da cui il `const`).

1 Realizzazione con primitiva e driver

Concentriamoci su una generica operazione di lettura, con interfaccia utente

¹Per vietare `in` e `out` sono necessari anche altri accorgimenti, che tralasciamo per semplicità; maggiori dettagli si trovano nel codice.

```
extern "C" void read_n(nat1 id, char* buf, natq quanti)
```

Assumiamo che nel sistema siano installate diverse periferiche simili, identificate dunque dal parametro *id*. Assumiamo inoltre che queste periferiche siano in grado di trasferire un byte alla volta, inviando una richiesta di interruzione ogni volta che è disponibile un nuovo byte. L'interfaccia di ogni periferica avrà un registro di controllo per abilitare e disabilitare le interruzioni (CTL) e un registro di ingresso da cui leggere il nuovo byte (RBR). La lettura da RBR funziona da risposta alla richiesta di interruzione: l'interfaccia non genera una nuova richiesta di interruzione fino a quando non ha avuto una risposta a quella precedente.

L'operazione sarà svolta in parte dalla primitiva e in parte da un *driver*, che andrà in esecuzione ad ogni richiesta di interruzione da parte dell'interfaccia:

- la primitiva ha lo scopo di avviare l'operazione di I/O e bloccare il processo, garantendo anche la mutua esclusione;
- il driver ha il compito di trasferire effettivamente i byte e sbloccare il processo quando l'operazione si è conclusa.

Per gestire la mutua esclusione e la sincronizzazione vogliamo usare i semafori, ma questo comporta che la primitiva non può essere atomica, e dunque non deve salvare e caricare lo stato del processo all'entrata e all'uscita dal sistema. In questo caso ha senso immaginare che sia il processo stesso ad eseguire la primitiva, sospendendosi e risvegliandosi al suo interno in corrispondenza delle invocazioni delle primitive semaforiche.

Consideriamo ora un processo P_1 che invoca la primitiva `read_n`. Questa, come per tutte le primitive, è in realtà solo una piccola funzione scritta in Assembler nel file `utente.s`. La funzione invoca la primitiva vera e propria tramite una istruzione `int`, che permette l'innalzamento del livello di privilegio:

```
1 .global read_n
2 read_n:
3     int $IO_TIPO_RN
4     ret
```

Nell'entrata corrispondente al tipo `IO_TIPO_RN` della tabella IDT dovrà essere installato un gate con che punti a `a_read_n`. Questa sarà una funzione scritta in assembler nel file `sistema.s`:

```
1 .extern c_read_n
2 a_read_n:
3     call c_read_n
4     iretq
```

Si noti che, come dicevamo, la `a_read_n` non salva lo stato, lasciando che la primitiva venga eseguita nel contesto del processo P_1 che l'ha invocata.

Passiamo ora alla parte C++ della primitiva. Dato l'identificatore `id`, la primitiva ha bisogno di ottenere alcune informazioni sulla corrispondente periferica (l'indirizzo dei suoi registri, ma non solo). Per memorizzare tutte queste informazioni prevediamo un descrittore di operazione di I/O definito come segue:

```

1  struct des_io {
2      ioaddr iRBR, iCTL;
3      char* buf;
4      natq quanti;
5      natl mutex;
6      natl sync;
7  };

```

È sufficiente avere un array di tali descrittori e usare `id` come indice al suo interno. Ogni descrittore contiene tre tipi di informazioni: gli indirizzi dei registri (riga 2), le informazioni (ricevute dall'utente) su dove i byte vanno trasferiti (righe 3–4) e due identificatori di semafori (righe 5–6). Per realizzare la sincronizzazione e la mutua esclusione, come abbiamo anticipato, la `c_read_n` utilizzerà i semafori, che servono proprio a questo. Il semaforo `mutex` è inizializzato a 1 all'avvio del sistema e serve a garantire la mutua esclusione tra i processi che vogliono usare la periferica `id`. Il semaforo `sync` è inizializzato a 0 all'avvio del sistema e serve a realizzare la sincronizzazione tra il processo che ha richiesto l'operazione e la conclusione dell'operazione stessa.

La `c_read_n` è strutturata nel modo seguente:

```

1  extern "C" void c_read_n(natl id, natb *buf, natl quanti)
2  {
3      // controllo sui parametri
4      des_io *d = &array_des_io[id];
5
6      sem_wait(d->mutex);
7      d->buf = buf;
8      d->quanti = quanti;
9      outputb(1, d->iCTL);
10     sem_wait(d->sync);
11     sem_signal(d->mutex);
12 }

```

Alla riga 4 ottiene un puntatore al descrittore della interfaccia, per comodità di scrittura. Le righe 6 e 11 garantiscono la mutua esclusione: solo un processo alla volta può eseguire le righe 7–10.

Le righe 7 e 8 trasferiscono le informazioni sul buffer dell'utente all'interno del descrittore. Da qui le leggerà il driver quando andrà in esecuzione.

La riga 9 abilita le interruzioni (supponiamo che sia sufficiente scrivere 1 nel registro CTL). Da questo momento l'interfaccia può inviare richieste di interruzione ogni volta che ha un nuovo byte da trasferire. Si noti che per il momento le interruzioni esterne sono mascherate, in quanto ci troviamo nel modulo sistema.

La riga 10 blocca P_1 sul semaforo di sincronizzazione. A questo punto il sistema può passare ad eseguire altri processi. Mentre sono in esecuzione questi altri processi le interruzioni sono abilitate. Si noti che P_1 è bloccato all'interno della `sem_wait` alla riga 10 e dunque ancora dentro la zona di mutua esclusione. Fino a quando P_1 non verrà sbloccato e non eseguirà la `sem_signal(d->mutex)` alla riga 11 l'interfaccia non potrà essere usata da altri processi, come volevamo. Se un altro processo (andato in esecuzione in seguito al blocco di P_1) invocherà la `read_n` sulla stessa interfaccia, arriverà alla riga 6 e si bloccherà.

Passiamo ora ad esaminare il driver. Il driver andrà in esecuzione per effetto di una richiesta di interruzione da parte dell'interfaccia (richiesta che arriverà alla CPU tramite l'APIC). Il driver gira a livello sistema e, per poter tornare a livello utente, deve terminare anch'esso con una `iretq`. Anche il driver, dunque, avrà una parte scritta in assembler:

```

1 .extern c_driver
2 a_driver_i:
3     call salva_stato
4     movq $i, %rdi
5     call c_driver
6     call apic_send_EOI
7     call carica_stato
8     iretq

```

Per fare in modo che `a_driver_i` vada in esecuzione ogni volta che l'interfaccia genera una interruzione, occorre conoscere il tipo dell'interruzione generata e preparare il corrispondente gate della IDT in modo che punti a `a_driver_i`.

Chiamiamo P_2 il processo in esecuzione all'arrivo dell'interruzione.

Il driver salva e ripristina lo stato di P_2 (linee 3 e 7) perché può dover cambiare il processo in esecuzione. Infatti, quanto l'ultimo byte è stato trasferito, il driver deve risvegliare P_1 (che ora è bloccato nella `sem_signal(d->sync)` dentro `c_read_n`). Se P_1 ha una priorità maggiore di P_2 , è P_1 che deve andare in esecuzione, mentre P_2 deve tornare in coda pronti. La `carica_stato` alla riga 7 quindi, carica lo stato di P_2 o di P_1 , a seconda dei casi.

Alle righe 4-5 si chiama il driver vero e proprio, `c_driver`, scritto in C++. Dal momento che stiamo assumendo di trattare interfacce simili, `c_driver` può essere una funzione generica e ricevere un parametro che gli indichi l'interfaccia da gestire (linea 4). Si noti che il parametro che `a_driver_i` passa a `c_driver` è una costante. Questo perché ogni interfaccia genera una richiesta di interruzione di tipo diverso, quindi è sufficiente associare ad ogni tipo di interruzione una diversa copia di `a_driver_i`, ciascuna con una costante diversa.

Alla riga 6 inviamo l'End Of Interrupt al controllore APIC, in modo che lasci passare le interruzioni a priorità minore o uguale di quella appena gestita dal driver.

Si noti che il driver usa le risorse di P_2 . In particolare usa la sua pila sistema. Usa anche la memoria virtuale di P_2 , dal momento che non stiamo cambiando

il valore di **cr3**.

Vediamo ora il codice di **c_driver**:

```
1  extern "C" void c_driver(natl id)
2  {
3      des_io *d = &array_des_io[id];
4
5      d->quanti--;
6      if (d->quanti == 0) {
7          outputb(0, d->iCTL);
8          des_sem *s = &array_dess[sem];
9          s->counter++;
10
11         if (s->counter <= 0) {
12             des_proc* lavoro = rimozione_lista(s->pointer);
13             inspronti(); // preemption
14             inserimento_lista(pronti, lavoro);
15             schedulatore(); // preemption
16         }
17     }
18     char c = inputb(d->iRBR);
19     *d->buf = c;
20     d->buf++;
21 }
```

Alla riga 3 il driver ottiene un puntatore al descrittore della interfaccia, per comodità di scrittura.

Lo scopo principale del driver è leggere il nuovo byte dall'interfaccia e copiarlo nel buffer dell'utente, cosa che viene fatta alle righe 18-19. Alla riga 20 il puntatore al buffer viene incrementato, in modo che il prossimo byte venga copiato nella locazione successiva.

Alla riga 5 il driver decrementa **d->quanti**, che così contiene sempre il numero di byte ancora da leggere. Se **d->quanti** è arrivato a zero il byte letto alla riga 18 è l'ultimo byte richiesto dall'utente. Si deve quindi disabilitare l'interfaccia a generare interruzioni (riga 7) e risvegliare il processo che aveva iniziato l'operazione (righe 8-16).

Ci sono alcune cose da notare:

1. la disabilitazione delle interruzioni (riga 7) è eseguita *prima* della lettura del byte (riga 18);
2. invece di chiamare **sem_signal()** ripetiamo la parte centrale del suo codice (riga 8-16);
3. la scrittura in **d->buf** (riga 19) è eseguita mentre è attiva la memoria virtuale di P_2 , anche se il buffer era stato allocato da P_1 .

Il punto 1 è una conseguenza del fatto che la lettura del byte fa da risposta alla richiesta di interruzione da parte dell'interfaccia, che a quel punto può generarne un'altra se ha un nuovo byte disponibile. Quindi, se leggiamo l'ultimo byte mentre le interruzioni sono abilitate, è possibile che l'interfaccia generi una nuova interruzione, rimandando in esecuzione il driver, anche se nessun processo ha iniziato una operazione di lettura. Il driver leggerebbe questo byte e lo copierebbe dove punta `d->buf`, andando a sovrascrivere parti casuali della memoria.

Il punto 2 è una conseguenza del fatto che `sem_signal()` salva e ripristina lo stato, ma il driver non è un processo e non ha un suo descrittore di processo. In particolare, se `c_driver` chiamasse `sem_signal()` salverebbe lo stato del driver nel descrittore processo attivo, che è P_2 . Per di più sovrascriverebbe lo stato salvato alla riga 3 di `a_driver_i` (avremmo violato la regola di non avere due `salva_stato` senza una `carica_stato` in mezzo).

Dal momento che il driver fa parte del modulo sistema, potremmo pensare di chiamare direttamente `c_sem_signal()`, evitando il salvataggio e caricamento dello stato. Purtroppo, però, il controllo sulla validità del semaforo (`sem_valido()`) fallirebbe, in quanto `liv_chiamante()` assume che `c_sem_signal()` sia stata invocata per effetto di una istruzione `int`. Nel nostro caso, invece, `liv_chiamante()` accederebbe alla pila sistema di P_2 e scambierebbe il livello di P_2 , che molto probabilmente è utente, per il livello del chiamante della `c_sem_signal()`; dal momento che il semaforo `ce->sync` era stato allocato da livello sistema, segnalerebbe un errore. Non ci resta dunque che ricopiare il codice della `c_sem_signal()` escludendo i controlli dei parametri (o definire una ulteriore funzione che contenta solo tale codice). Si noti che il driver, in questo modo, manipola le code dei processi, e dunque deve essere eseguito con le interruzioni disabilitate. In altre parole, il driver deve essere atomico. Questo comporta che anche le richieste di interruzione a precedenza maggiore dovranno aspettare che il driver termini, prima di poter essere gestite.

Il punto 3 è un problema se P_1 ha allocato il buffer nella sua sezione utente/privata, dal momento che gli indirizzi virtuali delle sezioni private hanno significati diversi per ogni processo. In questo caso `c_driver` scriverebbe nella sezione utente/privata di P_2 , non di P_1 , causando un doppio problema: P_1 non riceverebbe i suoi dati e P_2 si vedrebbe sovrascrivere parti casuali della sua memoria. Dobbiamo quindi richiedere che i buffer per l'I/O vengano sempre allocati nella parte utente/condivisa. Dal punto di vista del linguaggio C++, nel nostro caso, questo comporta che i buffer devono essere dichiarati globali o allocati nello heap, e mai dichiarati come variabili locali. Questo perché le variabili locali vengono allocate in pila e abbiamo deciso che le pile si trovano in parti private. Nei sistemi che bloccano i processi che causano page fault, il buffer deve anche essere residente (non rimpiazzabile), in quanto il driver, non essendo un processo, non può essere bloccato.

Esaminiamo infine come devono essere impostati i campi del gate che porta a `a_read_n` e di quello che porta a `a_driver_i`. Per il primo avremo:

- il campo DPL (Descriptor Privilege Level) che indica che il gate può essere utilizzato da livello utente;
- il campo L (Level) che indica che, dopo il salto, il processore si deve trovare a livello sistema;
- il campo I/T che può indifferentemente indicare “Interrupt” o “Trap”, dal momento che la primitiva non manipola direttamente le code e i descrittori dei processi, ma noi assumeremo “Interrupt” per uniformità con il resto del codice del modulo sistema.

Per il gate della `a_driver_i` avremo:

- il campo DPL (Descriptor Privilege Level) che indica che il gate può essere utilizzato solo da livello sistema;
- il campo L (Level) che indica che, dopo il salto, il processore si deve trovare a livello sistema;
- il campo I/T che indica “Interrupt”, per quanto detto prima.

L’impostazione del campo DPL deriva da questa constatazione: tutti i gate della IDT possono essere usati indifferentemente da tutti e tre i meccanismi di interruzione (interruzioni esterne, eccezioni e istruzione `int`). Impostando DPL a livello sistema impediamo all’utente di invocare il driver in modo spurio, tramite una `int`.

1.1 Controllo sui parametri

Alla riga 3 della `c_read_n()` abbiamo lasciato un commento in cui si dice che è necessario controllare i parametri. Questo è sempre vero per tutte le primitive, in quanto i parametri arrivano dal livello utente che, per definizione, non è fidato. In particolare, prima di usare `id` alla riga 4 dobbiamo assicurarci che `id` sia un valido identificatore di periferica. Se abbiamo raggruppato tutti i descrittori all’inizio dell’array è sufficiente ricordare l’indice dell’ultimo e confrontarlo con `id`.

Si noti che, invece, non c’è bisogno di controllare che `id` sia valido alla riga 3 di `c_driver_i`, in quanto in quel caso `id` è stato passato da `a_driver_i` che, facendo parte del modulo sistema, è fidata.

In `c_read_n()` è molto importante controllare i parametri `buf` e `quanti`, per evitare il cosiddetto problema del *Cavallo di Troia*. L’utente, invece di passare l’indirizzo di un buffer che ha correttamente allocato, potrebbe passare (usando dei cast o scrivendo direttamente in assembler) l’indirizzo di qualunque cosa, anche di parti della memoria che appartengono al sistema o ad altri processi. I controlli di protezione eseguiti dalla CPU e dalla MMU sono inefficaci in questo caso: il passaggio dell’indirizzo “cattivo” alla primitiva comporta solo la scrittura di un numero in un registro e la CPU non controlla alcunché, in quanto non può conoscere lo scopo di questa operazione. L’apparentemente innocuo Cavallo di Troia attraversa dunque le mura del sistema. Quando,

successivamente, il sistema tenta di usare l'indirizzo per leggere o scrivere (nel nostro caso ciò accade alla riga 19 di `c_driver_i`), ecco che il cavallo si apre, in quanto la primitiva gira a livello sistema e dunque anche la MMU non esegue alcun controllo. Se non prendiamo provvedimenti, l'utente riesce a costringere il sistema a scrivere su (o leggere da) una zona di memoria a cui lui, normalmente, non avrebbe accesso.

Dobbiamo anche controllare che il buffer che l'utente ci passa non contenga indirizzi che non sono mappati, o non sono normalizzati, in quanto questi causerebbero una eccezione nel driver alla riga 19 ma abbiamo detto che il driver deve essere atomico e, dunque, non deve generare eccezioni.

Il modulo sistema mette a disposizione seguente funzione che può essere usata in questi casi:

```
bool c_access(vaddr begin, natq dim,
               bool writeable, bool shared);
```

La funzione controlla che l'intervallo [begin, begin+dim) non attraversi il massimo indirizzo (non ci sia un *wrap around*) e non contenga indirizzi che fanno parte del “buco” (e dunque non sono normalizzati). Inoltre, percorre l'albero di traduzione e controlla che tutti gli indirizzi dell'intervallo siano mappati, accessibili da livello utente e, se `writeable` è **true**, anche scrivibili. Inoltre, se `shared` è **true**, controlla anche che tutto l'intervallo sia contenuto nella zona utente/condivisa.

La funzione `c_read_n()`, dunque, può eseguire il seguente codice

```
if (!c_access(begin, quanti, true)) {
    flog(LOG_WARN, "buf non valido");
    abort_p();
}
```

Si noti che passiamo **true** come parametro `writeable`, perché il driver dovrà *scrivere* nel buffer (in una `c_write_n()` avremmo passato **false**). Inoltre, dal momento che ci troviamo in una primitiva non atomica (che dunque non salva e carica lo stato), non è sufficiente chiamare `c_abort_p()`; per abortire il processo, ma si deve chiamare `abort_p()`, che salva lo stato, chiama `c_abort_p()` e poi carica lo stato.

DMA e PCI Bus Mastering

G. Lettieri

16 Marzo 2022

1 Direct Memory Access

Abbiamo visto, fino ad ora, due modalità di trasferimento dati da un dispositivo alla memoria, o viceversa:

- a “controllo di programma”;
- tramite interruzioni.

Nella prima modalità il software controlla periodicamente che il dispositivo sia pronto leggendo un qualche registro di stato, quindi opera il trasferimento con una o più operazioni di lettura da registri del dispositivo, seguite da scritture in memoria, o viceversa. Nella modalità con interruzioni il trasferimento avviene nello stesso modo, ma è iniziato solo quando il dispositivo segnala la propria disponibilità tramite una richiesta di interruzione. La modalità a controllo di programma è più veloce di quella a interruzioni, ma, come sappiamo, è complicata da programmare se si vuole che il processore possa fare anche altro nei momenti in cui il dispositivo non è pronto; entrambe le modalità prevedono comunque un coinvolgimento del processore, che deve eseguire le istruzioni di lettura e scrittura, e comportano che i dati vengano scambiati sul bus due volte: una volta tra RAM e CPU e una volta tra CPU e dispositivo.

La modalità DMA (Direct Memory Access), che è la terza e ultima modalità di trasferimento dati, prevede invece che sia direttamente il dispositivo ad eseguire le necessarie operazioni di lettura o scrittura sulla RAM, una volta istruito dal software. In generale, il software vorrà eseguire un trasferimento dati dal dispositivo verso un buffer in RAM (operazione di ingresso o lettura) o da un buffer in RAM verso un dispositivo (operazione di uscita o scrittura). Supponiamo che il buffer si trovi all’indirizzo b e sia grande n byte, occupando dunque gli indirizzi $[b, b + n]$. L’indirizzo b e il numero n devono essere comunicati al dispositivo, insieme a tutte le altre eventuali informazioni necessarie a definire il trasferimento richiesto. Da quel momento in poi il dispositivo si preoccuperà di eseguire autonomamente le operazioni in RAM, al proprio ritmo. Ovviamente il dispositivo deve essere dotato di un sommatore che gli permetta di calcolare da solo gli indirizzi necessari a partire da b , e di un contatore che decrementi n ogni volta che è stato completato un trasferimento. Quando tutti i byte sono stati

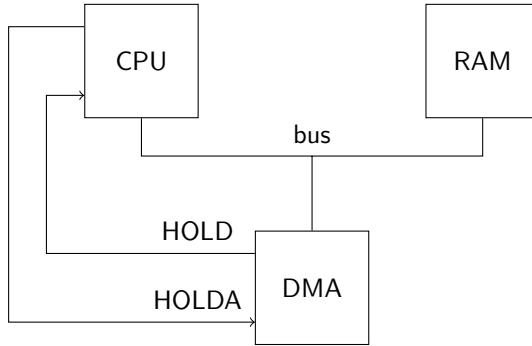


Figura 1: DMA, architettura di base.

trasferiti, il dispositivo segnalerà il completamento dell'operazione settando opportunamente un suo registro di stato e, tipicamente, inviando una richiesta di interruzione.

Dal punto di vista hardware possiamo considerare l'architettura schematizzata in Figura 1, dove abbiamo per il momento eliminato cache, MMU e bus PCI, ricreando l'architettura tipica dei primi PC IBM, o dei minicomputer PDP della DEC. Il dispositivo in grado di operare in DMA è collegato direttamente al bus dove si trovano anche la CPU e la RAM. Questo gli permette di dialogare con la RAM usando lo stesso protocollo già usato dalla CPU. Dal momento che il bus è condiviso, però, un solo dispositivo alla volta può pilotarlo: o la CPU, o il dispositivo DMA. L'accesso è arbitrato tramite i collegamenti HOLD/HOLDA fra dispositivo e CPU:

1. il dispositivo mantiene normalmente i suoi piedini di uscita in alta impedenza;
2. ogni volta che il dispositivo vuole eseguire un trasferimento sul bus, addiva HOLD;
3. in risposta, la CPU termina l'eventuale trasferimento in corso (che può essere anche nel mezzo di una istruzione), mette i suoi piedini di uscita in alta impedenza e attiva HOLDA;
4. il dispositivo attiva i suoi piedini di uscita ed esegue il trasferimento, quindi rimette le uscite in alta impedenza e disattiva HOLD;
5. la CPU disattiva HOLDA, attiva i suoi piedini e riprende il suo normale funzionamento.

In pratica, la CPU dà la precedenza al DMA nell'accesso al bus. La tecnica è chiamata “cycle stealing”, in quanto il DMA ruba cicli di bus alla CPU. Si noti che, mentre è in corso il trasferimento in DMA, la CPU è comunque in grado di eseguire una eventuale istruzione già prelevata, che non preveda accessi in memoria.

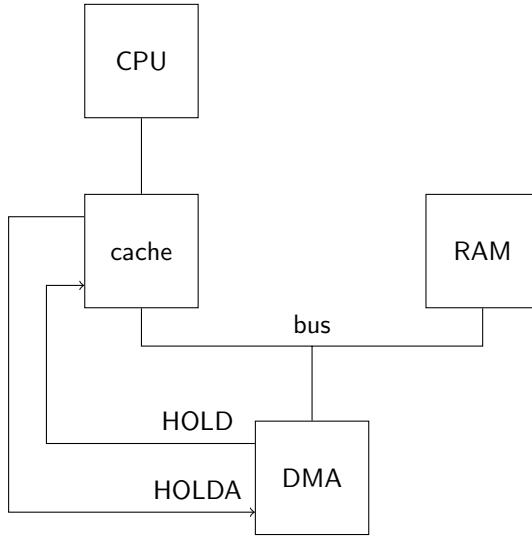


Figura 2: Interazione tra DMA e cache della CPU.

Il meccanismo HOLD/HOLDA funziona se c’è un unico dispositivo in grado di operare in DMA. Ci limitiamo a questo caso perché, come vedremo, il bus PCI ci permette di aggirare il problema senza introdurre altri meccanismi.

1.1 Interazione con la cache

Ora prendiamo in considerazione l’esistenza della cache. Notiamo subito che, come mostrato in Figura 2, i segnali HOLD e HOLDA devono ora collegare dispositivo DMA e controllore cache, in quanto è quest’ultimo, e non la CPU, ad essere collegato direttamente al bus con la memoria. A parte questo, l’handshake per il possesso del bus resta lo stesso.

L’introduzione della cache porta un grande vantaggio, ma anche alcune complicazioni. Il vantaggio è che è molto più probabile che la CPU riesca ad eseguire istruzioni mentre è in corso una operazione di DMA, perché può trovare istruzioni e operandi in cache.

Le complicazioni nascono dal fatto che le operazioni in DMA potrebbero coinvolgere parti di RAM che erano state precedentemente copiate in cache. Nel caso di cache con politica *write-back* la cache potrebbe anche contenere modifiche che non sono ancora state ricopiate in RAM. Esaminiamo i problemi, e le possibili soluzioni, partendo dal caso più semplice e passando poi a quelli più complessi. In ogni caso assumiamo che il software, per tutta la durata del trasferimento, non acceda al buffer coinvolto nel trasferimento. Ci interessa soltanto, dunque, lo stato della cache al momento di inizio e di fine del trasferimento.

1.1.1 Cache con politica *write-through*

Questo è il caso più semplice, perché all'inizio del trasferimento tutte le cacheline eventualmente presenti in cache contengono lo stesso valore delle corrispondenti cacheline in RAM. Questo implica che non ci sono problemi nel caso di operazione di uscita in DMA (direzione da RAM a dispositivo). Nel caso di operazione di ingresso in DMA (direzione da dispositivo a RAM), invece, bisogna assicurarsi che tutte le cacheline coinvolte (anche parzialmente) nel trasferimento vengano o rimosse dalla cache, o aggiornate, altrimenti il contenuto della cache, a fine trasferimento, non sarebbe consistente con il contenuto della RAM. Il software, dunque, potrebbe leggere valori “vecchi” scambiandoli per nuovi. Il problema è che l'accesso diretto alla memoria rende non più vera l'assunzione su cui abbiamo basato il funzionamento della cache, cioè che la RAM non cambia valore se non per le scritture che arrivano dalla CPU.

Il problema può essere risolto automaticamente in hardware, oppure essere demandato al programmatore software. Nel caso di soluzione completamente hardware, dobbiamo fare in modo che il controllore cache osservi *tutte* le possibili sorgenti di scritture in RAM. Grazie al bus condiviso, possiamo sfruttare il fatto che il controllore cache può osservare cosa sta accadendo durante l'accesso diretto alla RAM, ricevendo in ingresso le linee di controllo e di indirizzo. Se le linee di controllo identificano una operazione di scrittura, può usare il contenuto delle linee di indirizzo per eseguire una normale ricerca all'interno della cache (del tutto analoga a quella eseguita quando è la CPU a chiedere un indirizzo). Nel caso di *hit*, il controllore può autonomamente provvedere a invalidare la corrispondente cacheline. Alla fine del trasferimento, se il software accede al buffer, i dati dovranno essere prelevati dalla RAM, che contiene i valori aggiornati. Se il controllore riceve in ingresso anche le linee dati può aggiornare la cacheline, invece di invalidarla. La soluzione hardware appena illustrata viene detta *snooping* (ficcanasare), in quanto il controllore cache “ficca il naso” in quello che sta facendo il DMA.

Ci sono sistemi (per esempio quelli basati su processori ARM, comuni negli smartphone) in cui il problema non è risolto automaticamente in hardware e il programmatore deve risolverlo in software. Per farlo dispone di alcune istruzioni che gli permettono di interagire con il controllore cache, tipicamente per invalidare un intervallo di indirizzi. Il software deve eseguire queste istruzioni, specificando l'intervallo di indirizzi coinvolto nel trasferimento (nel nostro caso, gli indirizzi da b a $b + n$ escluso) subito prima che il trasferimento abbia inizio, o, equivalentemente, subito dopo che il trasferimento sia terminato.

1.1.2 Cache con politica *write-back* e trasferimento di intere cacheline

Consideriamo ora cache con politica *write-back*, ma assumiamo che ogni trasferimento in DMA coinvolga *interi* cacheline. Nel nostro caso stiamo assumendo che b sia allineato alla cacheline e n sia multiplo della dimensione di una cacheline.

Nel caso di cacheline non “dirty”, cioè che contengono in cache lo stesso valore che si trova nella corrispondente cacheline in RAM, ricadiamo negli stessi casi già visti e possiamo adottare soluzioni analoghe. Le cacheline dirty, invece, ci creano un nuovo problema nel caso di uscita in DMA (che, ricordiamo, implica una direzione da RAM a dispositivo). In questo caso la cache contiene la versione più aggiornata della cacheline che il DMA deve leggere, e sarebbe un errore se il DMA leggesse la versione contenuta in RAM. Anche questo problema può essere risolto in hardware con un meccanismo di snooping, ma non è sufficiente lo snooping da parte del controllore cache. Infatti, è vero che il controllore cache può accorgersi che l'operazione in DMA coinvolge una cacheline dirty, ma cosa dovrebbe fare una volta scoperto ciò? Pilotare le linee dati con il valore aggiornato della cacheline comporta una corsa, in quanto anche la RAM sta vedendo la stessa operazione sul bus, e anche lei vorrà pilotare le linee dati. Tra le varie possibili soluzioni hardware, quella tipicamente adottata in questo caso prevede che sia *il dispositivo DMA* a ficcanasare nello stato della cache, prima di iniziare l'operazione di lettura in RAM. Lo snooping può essere svolto su collegamenti dedicati tra dispositivo DMA e controllore cache, bypassando dunque la RAM, oppure acquisendo il controllo del bus con il normale protocollo HOLD/HOLD-A ed eseguendo una particolare operazione nota al controllore e ignorata dalla RAM (per esempio una operazione nello spazio di I/O). L'operazione di snooping deve passare al controllore cache l'indirizzo interessato dal trasferimento; il controllore deve eseguire una ricerca e, in caso di hit con cacheline dirty, può inviare lui stesso i dati al dispositivo, oppure riacquisire temporaneamente il controllo del bus e ricopiare la cacheline in RAM.

Nel caso di operazioni di ingresso in DMA (direzione da dispositivo a RAM) con cacheline dirty, c'è una ulteriore considerazione da fare. Questo caso può essere risolto con lo snooping da parte del controllore cache, come nel caso di cache write-through, ma il contenuto della cacheline dirty deve essere invalidato senza essere prima trasferito in RAM, in quanto il valore più aggiornato della cacheline deve essere quello prodotto dal dispositivo. Si noti che il controllore cache potrebbe iniziare il write back di questa cacheline (a causa di un rimpiazzamento) mentre è in corso il trasferimento DMA, ma il dispositivo non è ancora arrivato a trasferire la stessa cacheline, e dunque non ha fatto in tempo a invalidarla. Questo, per fortuna, non crea inconsistenze: la cache eseguirà il write-back, ma la cacheline in RAM verrà poi sovrascritta correttamente dal dispositivo.

Nel caso di soluzione interamente software, il programmatore può tipicamente ordinare al controllore cache di eseguire il write-back di un certo intervallo di indirizzi. Il software deve eseguire questa operazione su tutti gli indirizzi del buffer, e deve farlo necessariamente *prima* di avviare il trasferimento. Questo è ovviamente necessario per le operazioni di uscita (da RAM a dispositivo), in modo che la RAM sia correttamente aggiornata prima del trasferimento, ma è necessario anche per le operazioni di ingresso (da dispositivo a RAM) per evitare che eventuali cacheline dirty non vengano ricopiate in RAM dopo l'avvio dell'operazione, col rischio di sovrascrivere quanto già scritto dal dispositivo. Come ottimizzazione, il programmatore può disporre di una operazione che richiede

l'invalidazione *senza write-back* di un intervallo di indirizzi. Questa operazione può essere utilizzata prima di avviare un trasferimento in ingresso.

1.1.3 Cache con politica *write-back* e trasferimenti generici

Il caso più complicato è quello in cui il dispositivo deve scrivere in RAM parte di una cacheline, e questa si trova dirty in cache. La cacheline finale dovrà contenere il valore attualmente in cache per i byte non interessati dal trasferimento, e il valore scritto dal dispositivo per i rimanenti.

La soluzione interamente software cambia poco: il programmatore deve ordinare il write-back prima di avviare l'operazione, risolvendo il problema a monte. L'ottimizzazione dell'invalidazione senza write-back non può essere però usata, in quanto comporta la possibile perdita di dati nelle parti di memoria non appartenenti al buffer.

La soluzione interamente hardware può essere realizzata in vari modi, a seconda di quanto si vuol far fare al controllore cache e quanto al dispositivo. Il trasferimento in uscita (da RAM a dispositivo) non cambia sostanzialmente: il dispositivo è interessato solo a una parte della cacheline dirty, ma il contenuto da leggere è comunque interamente in cache, e si rende necessario uno snooping da parte del dispositivo. Il trasferimento in ingresso (da dispositivo a RAM), invece, può restare sostanzialmente immutato (con snooping da parte del controllore cache) solo se il controllore cache è in grado di aggiornare autonomamente la cache con i nuovi dati scritti dal dispositivo sul bus. Se, invece, il controllore è solo in grado di invalidare, il problema deve nuovamente essere risolto tramite snooping da parte del dispositivo: prima di eseguire il trasferimento parziale, il dispositivo esegue una operazione di snooping verso il controllore. In caso di hit, se il controllore risponde con un write-back, è sufficiente completare la normale scrittura in RAM. Se invece il controllore risponde inviando al dispositivo il contenuto della cacheline, il dispositivo deve combinare i dati contenuti nella cacheline con quelli da scrivere in memoria, e poi scrivere lui stesso l'intera cacheline in RAM.

In generale, un intero trasferimento che coinvolge un buffer $[b, b + n]$ sarà composto, in base all'allineamento di b e al valore di n , da un possibile trasferimento di parte di una cacheline, seguito da zero o più trasferimenti di intere cacheline, e concluso da un eventuale trasferimento di parte di una cacheline. Si noti che lo snooping da parte del controllore cache è più efficiente dello snooping da parte del dispositivo, in quanto il primo avviene in parallelo con l'operazione in RAM, mentre il secondo deve essere completato prima di poter dialogare con la RAM. I trasferimenti che coinvolgono intere cacheline possono essere realizzati più efficientemente con lo snooping da parte del controllore, e solo quelli alle estremità richiedono lo snooping da parte del dispositivo. Per ottimizzare i trasferimenti in ingresso è sufficiente che il dispositivo esegua il suo snooping solo nel caso di trasferimenti parziali, mentre il controllore cache esegua il suo in ogni caso.

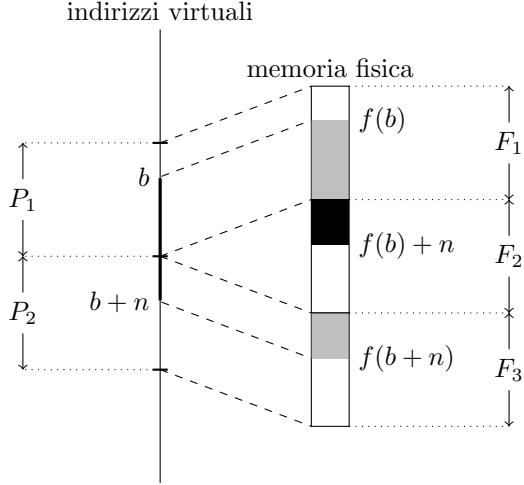


Figura 3: Un esempio con un buffer $[b, b + n]$ che attraversa due pagine (P_1 e P_2) mappate su due frame non contigui (F_1 e F_3).

1.2 Interazione con la memoria virtuale

Come sappiamo, tra la CPU e la cache troviamo la MMU, che traduce gli indirizzi virtuali (usati dal software) in indirizzi fisici (usati sul bus). Il dispositivo DMA, come collegato in Figura 2, non interagisce in alcun modo con la MMU e può utilizzare soltanto indirizzi fisici. Supponiamo che il software voglia eseguire un trasferimento in DMA da un buffer che si trova agli indirizzi *virtuali* $[b, b + n]$ verso un dispositivo (o viceversa). Saranno necessari i seguenti accorgimenti:

- al dispositivo andrà comunicato l'indirizzo *fisico* corrispondente a b , sia $f(b)$, e non l'indirizzo virtuale b ;
- se l'intervallo $[b, b + n]$ attraversa più pagine che non sono tradotte in frame contigui, il trasferimento deve essere spezzato in più trasferimenti in modo che ciascuno di essi coinvolga solo frame contigui;
- la traduzione di tutti gli indirizzi in $[b, b + n]$ non deve cambiare mentre il trasferimento è in corso.

Tutti e tre i punti sono diretta conseguenza del fatto che il dispositivo non ha accesso alla traduzione da indirizzi virtuali a fisici. Consideriamo il primo punto: se comunicassimo b al dispositivo, questo lo userebbe come se fosse un indirizzo fisico, andando a leggere o scrivere in parti della memoria che non c'entrano niente con il buffer (tranne nel caso particolare in cui $b = f(b)$). Consideriamo il secondo punto e supponiamo che $[b, b + n]$ attraversi due pagine, P_1 e P_2 , mappate su due frame non contigui, F_1 e F_3 (si veda la Figura 3). Se al dispositivo comunichiamo $f(b)$ ed n , questo leggerà (o scriverà) agli indirizzi *fisici* $[f(b), f(b) + n]$, invadendo dunque il frame F_2 , con effetti disastrosi. Il

trasferimento, invece, deve essere spezzato in due parti: una da $f(b)$ fino alla fine di F_1 , e uno dall'inizio di F_3 fino a $f(b + n)$ escluso. Alcuni dispositivi possono essere programmati per eseguire autonomamente più trasferimenti in successione, specificando in anticipo l'indirizzo e il numero di byte di ciascun trasferimento. Anche nei dispositivi che non offrono questa funzione, comunque, è sempre possibile programmare i diversi trasferimenti uno dopo l'altro in software.

Consideriamo invece il terzo punto, immaginando di trovarci in un sistema multiprocesso che realizzi, per esempio, lo swap-in/swap-out dei processi per poter eseguire più processi di quanti ne possano entrare in RAM. Supponiamo che un processo P_1 avvii un trasferimento in DMA verso un suo buffer privato e, mentre il trasferimento è in corso, il sistema decida di eseguire lo swap-out di P_1 per caricare un altro processo P_2 al suo posto. Il dispositivo DMA è ignaro del cambiamento e continuerà leggere o scrivere agli indirizzi fisici precedentemente occupati dal buffer di P_1 e ora occupati da parti della memoria di P_2 , di nuovo con effetti disastrosi.

2 PCI Bus Mastering

La Figura 4 mostra un esempio di architettura con bus PCI. Come sappiamo, sul bus PCI ogni dispositivo (più preciamente, ogni funzione all'interno di ogni dispositivo) può comportarsi da “iniziatore” di una transazione. I dispositivi che sono in grado di essere iniziatori di transazioni sono detti *bus master*, e devono essere dotati dei collegamenti REQ/GNT verso l’arbitro, che ha il compito di coordinare l’accesso al bus tra i vari bus master. In Figura 4 abbiamo tre dispositivi collegati al bus PCI: il ponte, che è sempre bus master (in quanto deve iniziare le transazioni PCI per conto della CPU), un altro dispositivo bus master, e un terzo dispositivo che non è bus master. Solo il ponte e il secondo dispositivo hanno i collegamenti REQ/GNT con l’arbitro.

Dal punto di vista software l’operazione si svolge come se il dispositivo bus master fosse collegato direttamente al bus principale; in particolare, il software dialoga soltanto con il dispositivo per comunicargli, per esempio, l’indirizzo (fisico) del buffer il numero di byte da trasferire e la direzione del trasferimento. Dal punto di vista hardware, però, tutto il trasferimento avviene per tramite del ponte. Ogni volta che il dispositivo vuole iniziare un trasferimento da o verso la RAM, inizia una transazione di memoria sul bus PCI, all’indirizzo opportuno. Il ponte è configurato in modo da comportarsi come obiettivo per tutte le transazioni di memoria con indirizzi appartenenti alla RAM che si trova sul bus principale. Il ponte, quindi, risponde alla transazione iniziata dal bus master e, in caso di trasferimento da RAM a dispositivo, esegue le necessarie operazioni di lettura DMA sul bus principale; in caso di trasferimento dal dispositivo alla RAM, copia temporaneamente i dati in arrivo dal dispositivo in un buffer interno e, in parallelo, inizia le necessarie operazioni di scrittura in DMA verso la RAM. Sul bus principale il ponte si comporta esattamente come un normale dispositivo DMA, e per lui valgono tutte le considerazioni che abbiamo svolto

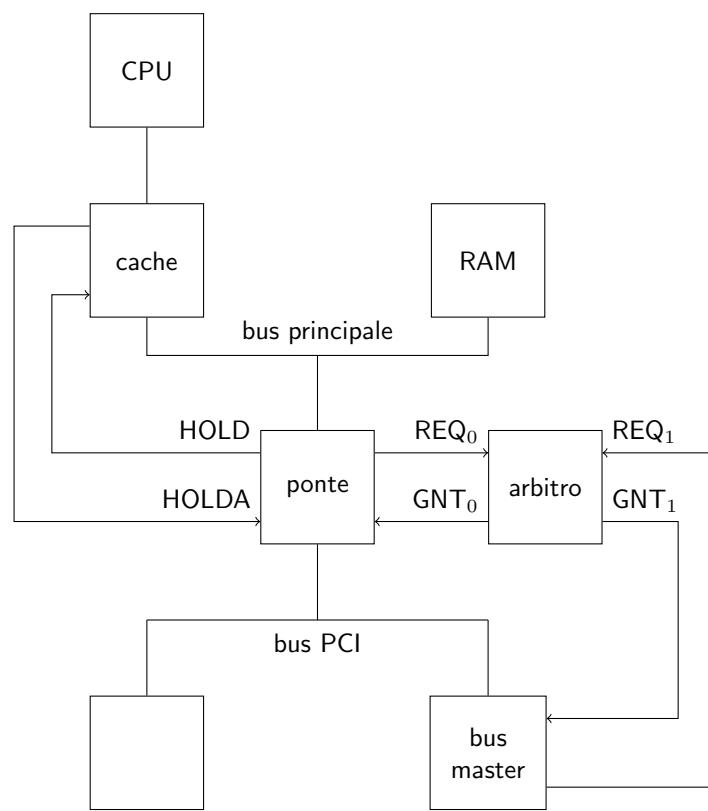


Figura 4: Bus Mastering PCI.

fino ad ora, in particolare per quanto riguarda il protocollo HOLD/HOLDA e le interazioni con la cache.

Si noti che il bus master può utilmente sfruttare la possibilità, offerta dallo standard PCI, di eseguire transazioni con più di una fase dati, in quanto conosce in anticipo il numero di byte da trasferire.

2.1 Interazione con il meccanismo delle interruzioni

La presenza della bufferizzazione intermedia nel ponte crea un problema con i trasferimenti in ingresso (da dispositivo a RAM) e il meccanismo delle interruzioni. In particolare, quando il dispositivo ha trasmesso al ponte l'ultimo byte richiesto, invierà una richiesta di interruzione per segnalare il completamento del trasferimento. Ci troviamo a questo punto di fronte ad una corsa: da una parte la richiesta di interruzione, che passa dal controllore delle interruzioni e arriva alla CPU, dall'altra i byte destinati alla memoria, che passano dal ponte: è possibile che la richiesta di interruzione arrivi e sia accettata prima che il ponte sia riuscito a trasferire tutti i byte in RAM; il software potrebbe quindi accedere erroneamente al buffer quanto ancora parte dei dati non sono stati aggiornati.

Anche questo problema può essere risolto in hardware o in software. Per risolverlo in software si può sfruttare il fatto che il ponte gestisce in modo strettamente FIFO tutti i trasferimenti di dati dal bus PCI verso il bus principale. La routine di interruzione, allora, prima di permettere l'accesso al buffer, potrebbe eseguire una lettura di un registro del dispositivo bus master. La risposta a questa lettura verrebbe accodata nel ponte *dopo* l'ultimo trasferimento di dati del bus master, e dunque l'istruzione di lettura verrebbe completata nella CPU necessariamente dopo che il ponte ha finito di trasferire i dati in RAM. Si noti che, dal momento che il dispositivo bus master aveva inviato una richiesta di interruzione, è probabilmente già previsto che la routine di interruzione debba leggere un qualche registro del dispositivo, per segnalare che la richiesta è stata servita: in tal caso, quest'unica lettura assolve entrambi i compiti.

Per risolvere il problema interamente in hardware, invece, si crea in genere un collegamento di handshake tra il controllore delle interruzioni e il ponte. Prima di inoltrare una qualunque richiesta di interruzione al processore, il controllore chiede l'OK al ponte; quest'ultimo non dà l'OK fino a quando non ha finito di trasferire tutti i dati ricevuti fino a quel momento. Un'altra soluzione, più moderna, prevede che le richieste di interruzione non viaggino su linee separate, ma siano inoltrate come speciali transazioni sul bus PCI stesso (Message Signaled Interrupts). In questo modo le richieste si accodano naturalmente ai dati e non ci sono problemi di corse.

Modulo I/O

G. Lettieri

21 Maggio 2018

La gestione delle interruzioni con il meccanismo del driver è poco flessibile ed efficiente, per due motivi:

- il driver deve essere eseguito con le interruzioni disabilitate, in quanto manipola direttamente le code dei processi;
- il driver non si può bloccare, in quanto non è un processo.

La disabilitazione delle interruzioni può causare problemi, in quanto costringe anche le interruzioni ad alta priorità ad aspettare che il driver termini la propria esecuzione. Il fatto che il driver non si possa bloccare limita fortemente le cose che può fare, soprattutto in un sistema più complicato in cui si debba gestire anche la rete o i file system.

Il secondo problema si può risolvere “trasformando” il driver in un processo. Più precisamente, facciamo in modo che l’interruzione non mandi in esecuzione l’intero driver, ma solo un piccolo *handler* che ha il solo scopo di mandare in esecuzione un processo, il quale si preoccuperà di svolgere le istruzioni che prima erano svolte dal driver.

Il problema delle interruzioni disabilitate può essere ridotto facendo girare questo processo (come tutti gli altri processi) a interruzioni abilitate, identificando tutti i punti in cui accede a strutture dati condivise (nel nostro caso, le code dei processi) e disabilitando le interruzioni solo in quei punti, usando le istruzioni `cli` e `sti`. Questa soluzione, per quanto utilizzata in sistemi reali, comporta però grandi complicazioni nella scrittura del codice. Possiamo invece adottare una soluzione molto più semplice: se il processo non fa parte del nucleo e appartiene invece ad un modulo distinto, che non è collegato con il modulo sistema, non ha modo di accedere alle code dei processi se non invocando delle primitive di sistema. Poiché le primitive di sistema girano a interruzioni disabilitate, ecco che l’accesso alle code dei processi è protetto.

Introduciamo allora un nuovo modulo, detto modulo *I/O*, distinto dal modulo sistema e dal modulo utente. Lo scopo di questo modulo è di realizzare le primitive per l’accesso alle periferiche, gestendo le richieste di interruzione tramite processi, detti processi “esterni” (nel senso che sono esterni al modulo sistema, anche se svolgono funzioni di sistema). Nella nostra implementazione, i file che contengono il codice di questo nuovo modulo si trovano nella cartella

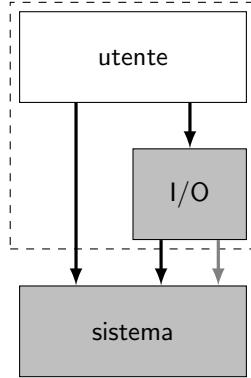


Figura 1: Relazioni tra i moduli. I moduli in grigio sono eseguiti con la CPU a livello sistema. I moduli dentro il riquadro tratteggiato sono eseguiti con le interruzioni mascherabili abilitate. Le frecce rappresentano possibili invocazioni di primitive: le nere sono accessibili anche da livello utente e le grigie solo da livello sistema.

io e sono `io.cpp` e `io.s`. La loro compilazione e collegamento produrrà il file `build/io`, che poi verrà caricato in memoria durante l'avvio del sistema e mappato nello spazio di indirizzamento di ogni processo, nella parte che abbiamo chiamato *IO/condivisa*. Il modulo `io` conterrà tutto ciò che è legato alle primitive di I/O, incluse le primitive di I/O invocabili dall'utente e le loro strutture dati.

La Figura 1 mostra le relazioni tra i moduli che compongono il sistema. Si noti che ora l'utente ha accesso sia a primitive che sono realizzate direttamente nel modulo sistema (`sem_ini()`, `sem_wait()`, `sem_signal()`, `delay()`, `activate_p()`, `terminate_p()`), sia a nuove primitive che sono realizzate nel modulo I/O. L'accesso a queste nuove primitive avviene sempre tramite la tabella IDT, con l'esecuzione di una istruzione `int`. Le relative entrate della tabella, però, punteranno a funzioni che sono definite nel modulo I/O.

Idealmente vorremmo che il codice contenuto in questo nuovo modulo girasse ad un livello intermedio tra utente e sistema in quanto deve avere più diritti degli utenti (deve poter interagire direttamente con le interfacce di I/O), ma non deve accedere direttamente alle strutture dati del sistema (IDT, GDT, code dei processi, tabelle della paginazione, etc.) Purtroppo il processore dispone di soli due livelli e scegiamo, dunque, di far girare anche questo modulo a livello sistema (in Figura, entrambi i moduli sono su sfondo grigio). Il fatto di compilare separatamente il modulo I/O e il modulo sistema protegge comunque le strutture dati del sistema da molti errori, ma sicuramente non da tutti (per esempio, un accesso errato in memoria da parte del codice I/O può comunque modificare la memoria riservata al modulo sistema).

A differenza del codice del modulo sistema, il codice del modulo I/O girerà a interruzioni abilitate, come il codice del modulo utente. Questo vale sia per

```

1 # file: sistema.s
2 handler_i:
3     call salva_stato
4     call inspronti
5     movq $i, %rcx
6     movq a_p(, %rcx, 8), %rax
7     movq %rax, esecuzione
8     call carica_stato
9     iretq

```

Figura 2: Schema generale di un handler.

il codice dei processi esterni, sia per il codice delle nuove primitive realizzate nel modulo I/O. Eventuali problemi di mutua esclusione dovranno essere risolti utilizzando i semafori forniti dal modulo sistema. Infatti, nella Figura si vede che anche il modulo I/O fa uso di primitive fornite dal modulo sistema; in particolare, usa le primitive semaforiche.

Il modulo I/O ha però accesso anche a primitive ad esso dedicate e non accessibili da livello utente. Per evitare che gli utenti possano invocare queste primitive è sufficiente che il bit DPL dei corrispondenti gate sia impostato a Sistema.

Una delle primitive riservate al modulo I/O è la primitiva `activate_pe()`, che serve ad attivare un processo esterno. Questa primitiva ha gli stessi parametri della normale `activate_p()`, con un ulteriore parametro che è il numero del piedino dell'APIC da cui arriveranno le richieste a cui il processo deve rispondere. Il modulo sistema avrà una tabella `a_p` con una entrata per ogni piedino dell'APIC. La `activate_pe()`, dopo aver creato il processo, inserirà il corrispondente `des_proc` nella opportuna entrata di questa tabella, invece di inserirlo in coda pronti.

Per ogni possibile interruzione, il modulo sistema deve predisporre un handler che si preoccupa di mettere in esecuzione il corrispondente processo esterno, prendendo il `des_proc` da questa tabella. Quindi, se i è uno dei piedini dell'APIC, dovrà esistere un `handler_i` che metta in esecuzione il `des_proc` preso da `a_p[i]`. Questi handler non sono inizialmente associati a nessuna entrata della IDT. Il motivo è che l'entrata della IDT, ovvero il tipo dell'interruzione, determina anche la priorità che l'APIC assegna alla richiesta di interruzione. Vogliamo fare in modo che questa priorità sia consistente con la precedenza del corrispondente processo esterno, come assegnata dalla `activate_pe()`. La soluzione adottata prevede che tale precedenza debba avere la forma `MIN_EXT_PRIO + prec`, dove `MIN_EXT_PRIO` è una costante definita dal sistema (in `costanti.h`) e `prec` è un numero naturale minore di 256. La `activate_pe()` programmerà l'APIC in modo che il piedino i invii il tipo `prio` e all'entrata `prio` della IDT installerà un gate che punti a `handler_i`.

```

1 // file: io.cpp
2 extern "C" estern(nat1 i)
3 {
4     des_io *d = &array_des_io[i];
5
6     for (;;) {
7         ...
8         wfi();
9     }
10 }
```

Figura 3: Schema generale di un processo esterno.

```

1 # file: io.s
2 .global wfi
3 wfi:
4     int $TIPO_WFI
5     ret
```

Figura 4: Funzione di interfaccia per la primitiva wfi().

L'handler associato alla richiesta di interruzione proveniente dal piedino i -esimo avrà la forma mostrata in Figura 2. Alla riga 3 salva lo stato del processo che stava girando quando la richiesta è stata accettata e alla riga 4 lo inserisce forzatamente in testa alla coda pronti (se era in esecuzione, era sicuramente quello a priorità maggiore tra quelli in coda pronti). Nelle righe 5-7 esegue l'equivalente del codice `esecuzione=a_p[i]`. La successiva coppia “**call** carica_stato; **iretq**” (righe 8 e 9) cederà il controllo al processo esterno.

I processi esterni seguiranno tutti lo schema generale mostrato in Figura 3. In Figura si assume che nel sistema ci siano più periferiche simili, ciascuna gestita da un diverso processo esterno, il cui codice può essere però scritto una volta per tutte. Tramite il parametro i il codice accede al descrittore di una specifica interfaccia (linea 4). Si noti che tale parametro era stato passato alla `activate_pe()` al momento della creazione del processo, in modo del tutto analogo a quanto avviene con la `activate_p()`. Dopo la loro creazione i processi esterni non terminano più e, dopo aver risposto ad una richiesta di interruzione, si limitano a sospendersi in attesa della prossima. Il loro corpo è composto dunque da un ciclo infinito (linee 6-9) che, dopo ogni iterazione, termina con una invocazione della primitiva di sistema `wfi()` (*wait for interrupt*), come si vede alla linea 8. Anche questa primitiva è riservata al modulo I/O.

La Figura 4 mostra il codice della funzione di interfaccia `wfi()`, usata dal modulo I/O per invocare la primitiva di sistema vera e propria, mostrata in Figura 5. La primitiva salva lo stato del processo esterno (linea 3), invia l'EOI

```

1 # file: sistema.s
2 a_wfi:
3     call salva_stato
4     call apic_send_EOI
5     call schedulatore
6     call carica_stato
7     iretq

```

Figura 5: La primitiva di sistema `wfi()`.

al controllore APIC (linea 4) e mette in esecuzione un altro processo (linee 5-7), di fatto sospendendo il processo esterno, che a questo punto potrà andare nuovamente in esecuzione solo quando il corrispondente handler verrà nuovamente invocato, in risposta ad una nuova richiesta di interruzione da parte della stessa interfaccia.

Si noti che possiamo affermare con certezza che, dopo la prima volta che il processo esterno è andato in esecuzione, la coppia “`call carica_stato; iretq`” al termine dell’handler associato (linee 6-7 di Figura 2) caricherà lo stato salvato alla linea 3 della `a_wfi`. Infatti, se l’handler è in esecuzione vuol dire che l’interfaccia ha inviato una richiesta che l’APIC ha fatto passare, e dunque l’APIC deve aver ricevuto l’EOI per la richiesta precedente, e dunque l’ultima cosa che il processo esterno aveva fatto in precedenza era proprio chiamare la `wfi()`.

Non possiamo invece sapere quale processo verrà messo in esecuzione al termine della `a_wfi`. Questo perché il processo esterno gira a interruzioni abilitate e dunque, mentre è stato in esecuzione, vari processi potrebbero essere finiti in coda pronti (per esempio, processi sospesi su una `delay()`, o processi che attendevano la conclusione di operazioni di I/O su altre periferiche a maggiore priorità, etc.).

1 Esempio di primitiva di lettura

Torniamo a considerare una generica operazione di lettura, con interfaccia utente

```
extern "C" void read_n(nat1 id, char* buf, natq quanti);
```

L’operazione sarà svolta in parte dalla primitiva e in parte da un processo esterno messo in esecuzione da un handler ad ogni richiesta di interruzione da parte dell’interfaccia.

Consideriamo ora un processo P_1 che invoca la primitiva `read_n()`. Questa, come per tutte le primitive, è in realtà solo una piccola funzione scritta in Assembler nel file `utente.s`. La funzione invoca la primitiva vera e propria tramite una istruzione `int`, che permette l’innalzamento del livello di privilegio:

```
1 # file: utente.s
```

```

2   .global read_n
3   read_n:
4       int $IO_TIPO_RN
5       ret

```

All'entrata `IO_TIPO_RN` della tabella IDT dovrà essere installato un gate con che punti a `a_read_n`. Questa sarà una funzione scritta in assembler nel file `io.s`:

```

1   # file: io.s
2   .extern c_read_n
3   a_read_n:
4       call c_read_n
5       iretq

```

Notiamo che la funzione `a_read_n` *non* chiama le funzioni `salva_stato` e `carica_stato`, esattamente come l'analogia nel caso del driver. Il motivo è lo stesso: la primitiva `read_n()` non è atomica ed è eseguita dal processo che la invoca. In questo caso, però, non c'è modo di chiamare `salva_stato` o `carica_stato` per sbaglio, in quanto si tratta di funzioni definite nel modulo sistema, che non è collegato con il modulo I/O.

Passiamo ora alla parte C++ della primitiva. Anche in questo caso la primitiva ha bisogno di ricordare alcune informazioni relative alla periferica, quindi prevediamo un descrittore di operazioni di I/O, identico a quello visto precedentemente:

```

1 // file: io.cpp
2 struct des_io {
3     natw iRBR, iCTL;
4     char* buf;
5     natl quanti;
6     natl mutex;
7     natl sync;
8 };

```

Prevederemo come al solito un array di tali descrittori e useremo `id` come indice al suo interno. In questo caso, però, l'array sarà definito in `io.cpp`.

Anche la `c_read_n` è identica a quella vista precedentemente, che riportiamo qui per comodità:

```

1 // file: io.cpp
2 extern "C"
3 void c_read_n(natl id, natb *buf, natl quanti)
4 {
5     des_io *d = &array_des_io[id];
6
7     sem_wait(d->mutex);
8     d->buf = buf;

```

```

9     d->quanti = quanti;
10    outputb(1, d->iCTL);
11    sem_wait(d->sync);
12    sem_signal(d->mutex);
13 }

```

C'è però una differenza rispetto al caso precedente, anche se non si vede: questa primitiva è ora definita nel modulo I/O (file `io.cpp`) e gira ad interruzioni abilitate (il suo gate è di tipo *trap*). Non ci sono comunque problemi di interferenza con altri processi che potrebbero interromperla, grazie alla mutua esclusione garantita dal semaforo `d->mutex`. Si noti come la mutua esclusione è rilasciata dopo aver atteso, sul semaforo `d->sync`, che il trasferimento sia interamente completato. Chiunque volesse utilizzare la periferica mentre è già un corso un altro trasferimento dovrà dunque mettersi in fila alla riga 7, aspettando che il trasferimento sia finito.

Vediamo ora il codice del processo esterno:

```

1 // file: io.cpp
2 extern "C" void estern(nat1 id)
3 {
4     des_io *d = &array_des_io[id];
5
6     for (;;) {
7         d->quanti--;
8         if (d->quanti == 0)
9             outputb(0, d->iCTL);
10        char c = inputb(d->iRBR);
11        *d->buf = c;
12        d->buf++;
13        if (d->quanti == 0)
14            sem_signal(d->sync);
15        wfi();
16    }
17 }

```

Lo scopo principale del processo esterno, come quello del driver, è di leggere il nuovo byte dall'interfaccia e copiarlo nel buffer dell'utente. Il codice del processo esterno è dunque molto simile a quello del driver, ma ci sono delle differenze dovute al fatto che ora ci troviamo in un vero processo. In particolare, il test su `d->quanti == 0` deve essere ripetuto due volte:

1. alla riga 8, in quanto dobbiamo eventualmente disabilitare le interruzioni (riga 9) *prima* di leggere il byte dall'interfaccia (riga 10);
2. alla riga 13, in quanto dobbiamo risvegliare il processo che aveva richiesto il trasferimento (riga 14) *dopo* aver effettivamente trasferito l'ultimo byte (righe 10-11).

Per il punto 1 vale esattamente lo stesso discorso già fatto nel caso del driver: se lasciassimo le interruzioni abilitate e leggessimo l'ultimo byte, l'interfaccia potrebbe generare una nuova richiesta, portando al trasferimento di un ulteriore byte che non era stato richiesto.

Il punto 2 è dovuto al fatto che, a differenza del driver, il processo esterno non è atomico. In particolare, la `sem_signal()` (linea 14) potrebbe mettere in esecuzione il processo risvegliato, se questo ha priorità maggiore del processo esterno. Il processo risvegliato (che era quello che aveva richiesto il trasferimento) potrebbe così cominciare ad usare i dati, prima che l'ultimo byte sia stato effettivamente trasferito.

La Figura 6 mostra un esempio di evoluzione del sistema supponendo che un processo P_1 invochi una `read_n()` chiedendo di trasferire 2 byte dall'interfaccia i . Si suppone che sia pronto anche un processo P_2 , a priorità più bassa, e che non ci siano altre richieste di interruzioni oltre quelle dell'interfaccia i durante tutto il trasferimento.

2 Interazioni con gli altri meccanismi

Vediamo ora come le operazioni di I/O interagiscono con il meccanismo della memoria virtuale e del Bus Mastering. Per quanto riguarda il Bus Mastering, la cosa a cui prestare attenzione è che i trasferimenti eseguiti dalle periferiche in grado di operare in questa modalità non attraversano la MMU, e dunque sono eseguiti ad indirizzi *fisici*. Per quanto riguarda la memoria virtuale, la cosa a cui prestare attenzione è che i byte sono trasferiti mentre in esecuzione c'è, in generale, un processo *diverso* da quello che aveva richiesto il trasferimento.

Pensiamo ora al fatto che il processo utente che ha iniziato il trasferimento ha passato alla `read_n()` l'indirizzo di un suo buffer. Il buffer si trova ovviamente nella memoria *virtuale* del processo. Restano vere tutte le limitazioni dovute al problema del Cavallo di Troia, che richiede che indirizzi del buffer accessibili da livello utente e correttamente mappati. Oltre a queste, però, ci sono ulteriori limitazioni, che dipendono dal modo in cui è realizzata il trasferimento.

Nel caso di primitiva di sistema con driver, teniamo presente che il driver userà lo spazio di indirizzamento del processo che ha interrotto. Se vogliamo che il driver possa accedere al buffer anche da questo spazio, il buffer dovrà trovarsi nella parte utente/condivisa, in modo che gli indirizzi del buffer abbiano lo stesso significato per tutti i processi. Nel nostro caso questo comporta che deve trovarsi nella sezioni `.data` o `.bss` o nello heap. Se il processo utente è scritto in C++, questo implica che il buffer dovrà essere dichiarato a livello globale o allocato tramite `new`.

Consideriamo ora il caso di primitiva di I/O con handler e processo esterno. Il buffer dovrà essere accessibile sia al processo utente, sia al processo esterno. Nel nostro caso questo comporta, di nuovo, che il buffer deve trovarsi nella parte utente/condivisa (sezioni `.data` o `.bss` o heap).

Pensiamo infine al caso del Bus Mastering. L'indirizzo del buffer passato alla primitiva non può essere direttamente utilizzato dalla periferica, ma va

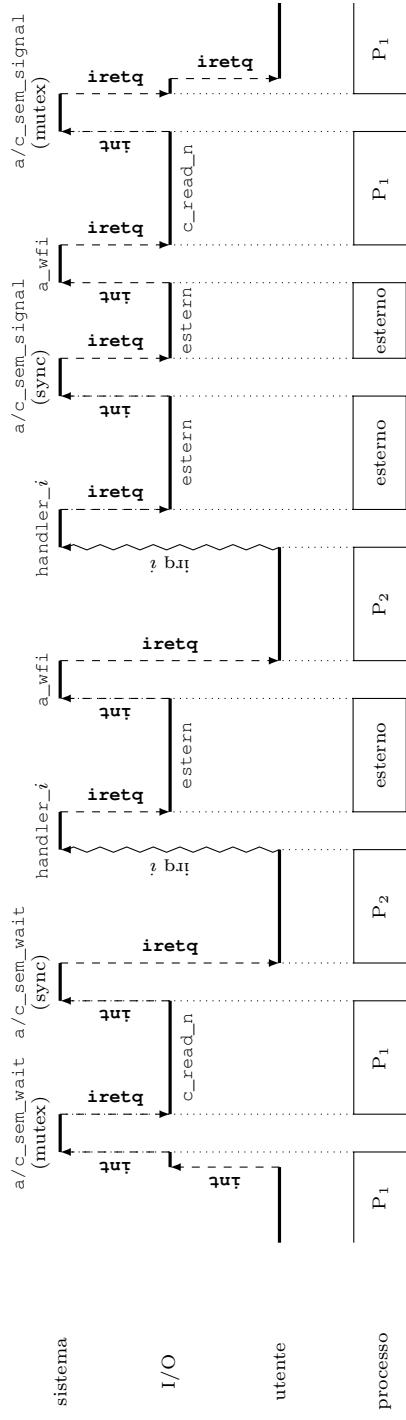


Figura 6: Esempio di esecuzione temporale in seguito all'invocazione di `read_n()` da parte di un processo P_1 (trasferimento di due byte). Le tre righe in alto mostrano il codice attualmente in esecuzione. La riga in basso mostra quale processo sta eseguendo il codice. In particolare, quando siamo nel modulo sistema tutti i processi sono fermi e il codice è eseguito atomicamente, mentre il codice del modulo I/O è sempre eseguito da un processo (quello che ha invocato la primitiva oppure un processo esterno).

prima tradotto in fisico. Dal momento che gli alberi di traduzione sono gestiti dal modulo sistema e non vogliamo che il modulo I/O vi acceda direttamente, forniamo al modulo I/O la seguente primitiva di sistema:

```
paddr trasforma(vaddr v);
```

Questa primitiva, dato un indirizzo virtuale v , restituisce il corrispondente indirizzo fisico nello spazio di indirizzamento del processo corrente (0 se l'indirizzo non è mappato). A differenza degli altri due meccanismi, questa volta non è necessario che il buffer si trovi nella parte utente/condivisa, proprio perché la periferica accederà direttamente alla memoria fisica, senza bisogno di traduzioni di indirizzi. Per lo stesso motivo, però, è fondamentale che il buffer non sia rimpiazzato dalla memoria (per esempio tramite uno swap-out del processo) per tutta la durata del trasferimento (la periferica non ha modo di sapere che la memoria fisica in cui sta scrivendo, o da cui sta leggendo, ora contiene qualcosa d'altro). Se il buffer attraversa più pagine, infine, non sarà in generale possibile completare tutto il trasferimento in un'unica operazione, in quanto i frame che contengono il buffer potrebbero non essere contigui in memoria.