Domande e risposte dalla mailing list

Dr. Ing. Giuseppe Lettieri

a cura di Riccardo Diodati

Indice

Parte I

Architettura dei calcolatori con riferimento al personal computer

Capitolo 1

Architettura, dispositivi e struttura dei programmi

Si può utilizzare il circuito del bus a 16 bit per accedere alla memoria? Che problemi si potrebbe riscontrare?

Sì (in presenza di un circuito come quello di Fig 7.12 del libro bianco). Ovviamente, non si potranno eseguire operazioni di lettura a 32 bit.

Non riesco ad immaginarmi come è fatta una memoria dinamica. Che vuol dire singolo piano di celle?

Che le *celle* (le parti elementari che memorizzano un singolo bit ciascuna) sono organizzate in una matrice quadrata su un unico piano. Le memorie con questa organizzazione sono $N \times 1$, nel senso che vanno pensate come N locazioni di un bit ciascuna (in altre parole, ogni bit ha il suo indirizzo specifico). Per avere memorie $N \times M$, si realizzano M piani uno sopra l'altro: ogni piano contiene una matrice di celle

Perché *W/R* va ad 1 dopo */RAS* ma prima di */CAS*?

Deve essere portato a 1 prima che */CAS* venga attivato, altrimenti la memoria penserebbe che si tratta di una operazione di lettura e porterebbe *DO* in conduzione (invece, vogliamo che resti in alta impedenza, per poter collegare lo stesso filo a *DI*). Possiamo anche portarlo a 1 prima di */RAS*, ma facendolo dopo otteniamo il vantaggio che la prima parte di ogni ciclo (sia di lettura che di scrittura) è la stessa. In particolare, la memoria farà sempre la stessa cosa quando vede */RAS* attivo: lettura (interna) di una intera riga della matrice, con conseguente rinfresco di tutte le celle della riga.

Il circuito di abilitazione è progettato in modo tale che, a seconda degli indirizzi generati dal processore, riesca a capire quale uscita abilitare?

Sì

A che serve */BHE*?

A capire, quando *A0*=*/BLE*=0, se se l'operazione di lettura/scrittura coinvolge un operando di 8 oppure 16 bit.

Non mi è molto chiara la temporizzazione raffigurata a pagina 136 del libro bianco, soprattutto in relazione alla descrizione data alla pagina precedente. Più che altro non mi convince il fatto che */READY*, dopo essersi stabilizzato a 1, torni a *oscillare* a cavallo del segnale di clock successivo, per poi stabilizzarsi a 0. Che senso ha? Dopo essersi stabilizzato a 1 non dovrebbe essere già stabile a 0 al fronte successivo? Altrimenti leggiamo un valore che oscilla.

Quel simbolismo non vuol dire che il segnale può oscillare, ma che il segnale può andare a zero in un istante qualunque (non ci sono delle `X`, ma il segnale che va a zero ripetutamente)

Nel libro si dice che le interfacce funzionanti con tipo di ciclo doppio possono essere posizionate su un qualunque bus dati con ampiezza uguale a quella dei dati da trasferire. Non andrebbe bene anche un bus dati con ampiezza maggiore di quella dei dati da trasferire?

Non ho con me il libro per controllare, ma la frase che lei riporta mi suona strana: o possono essere posizionate su un bus qualunque, o devono essere posizionate su quello che ha la stessa ampiezza dei dati da trasferire. In ogni caso, per quelle a ciclo doppio, va bene un bus qualunque (quindi, non importa la dimensione dei dati da trasferire).

A pagina 140 del libro bianco è riportata la temporizzazione dei segnali prodotti dai circuiti di abilitazione comando per i cicli di lettura e scrittura nello spazio di memoria. Ho un dubbio sul caso della lettura. Il segnale *D31_D0* è stabile prima che divenga attivo il segnale */MRD*. I banchi di memoria, non dovrebbero essere abilitati a trasmettere in uscita solo dopo la transizione a 0 di */MRD*? (a reti logiche abbiamo visto che ci sono delle porte 3-state che abilitano le uscite solo se */MRD* vale 0).

È come dice lei (anche qui non posso controllare il libro per cercare di capire le intenzioni del professore).

Durante il ciclo di risposta ad una richiesta di interruzione, il processore compie due accessi allo spazio di I/O, agli indirizzi 0 e 4. C'è un motivo particolare per cui sono stati scelti proprio quei due indirizzi?

Non che io sappia. Probabilmente, erano due indirizzi in cui non era normalmente montata nessuna interfaccia.

Salve professore sono un suo vecchio studente di calcolatori elettronici siccome sono molto incuriosito dalle architetture degli elaboratori (ho fatto anche l'esame di TIA) e sono affascinato dal linguaggio assembler, anche se magari non avrà tempo mi farebbe piacere avere da lei un elenco dei migliori libri per approfondire la materia.

Il libro che le consiglio di leggere ad ogni costo è:

"*Computer Architecture: A Quantitative Approach*" di John L. Hennessy e David A. Patterson, edito da Morgan Kaufmann

Gli autori sono gli inventori dell'archiettura RISC e il libro è adottato come testo in, credo, tutte le università del mondo. Recentemente ne è uscita la quarta edizione. Si parla diffusamente dei concetti più avanzati dell'architettura dei processori (dal pipelining in poi).

Dagli stessi autori, c'è anche, in italiano:

"*Struttura, organizzazione e progetto dei calcolatori - interdipendenza tra hardware e software*" edito da Jackson libri

Questo è però un testo più introduttivo, che copre grossomodo quanto si è fatto a Calcolatori Elettronici.

Con i libri precedenti si resta nell'ambito delle archietetture *classiche*. Per le architetture più avanzate, le consiglio, invece:

"*Parallel Computer Architecture: A Hardware/Software Approach*" di David E. Culler, Jaswinder Pal Singh, Anoop Gupta edito da Morgan Kaufmann

Infine, un argomento molto interessante e che le consiglio vivamente, è la storia delle architetture. Un buon punto di partenza è il sito:

```
http://ed-thelen.org/comp-hist/index.html
```

In teoria è possibile riscrivere un driver che gestisce un'interfaccia a 8 bit montata su bus dati a 8, in modo che funzioni correttamente anche se collegata ad un bus dati a 32?

Mah, se lo riscriviamo, perché non dovrebbe essere possibile?

Ipotizzando di non poter riscrivere nulla, l'incompatibilità è ambo i lati (cioè sia se si cerca di far girare qualcosa progettato a 8 bit su bus dati a 32, che viceversa), oppure vale solo in un senso?

L'incompatibilità c'è in ambo i versi, e il driver va riscritto. Attenzione che *riscrivere il driver* può anche voler dire, semplicemente, ricompilarlo dopo aver cambiato il valore di qualche costante: quello che succede è che, nel bus a 8 bit, le porte dell'interfaccia appaiono ad indirizzi consecutivi, nel bus a 32 bit appaiono a indirizzi che sono multipli di 4. Se, invece di usare delle costanti, usiamo delle variabili, possiamo anche scrivere un driver che funziona in entrambi i modi, senza la necessità di essere ricompilato. Il problema, del tutto pratico, che aveva costretto l'IBM a mantenere anche i bus a 8 e 16 bit, era che, purtroppo, i driver di molte periferiche non erano stati scritti in questo modo.

C'è qualcuno o qualcosa che controlla se sul bus dati a 16 bit si cerca di trasferire una parola non allineata?

Nel calcolatore semplificato che vediamo a lezione, no. Nel vero PC IBM da cui è tratto, la circuiteria di interfacciamento al bus provvedeva anche a riordinare i trasferimenti non allineati.

Il piedino *READY* che arriva al processore: chi si serve di questo pin per comunicare col processore?

Ad esempio il controllore cache, in caso di miss. Nello spazio di I/O, può essere pilotato da eventuali dispositivi più lenti degli altri.

Non è che i banchi di memoria devono avere un'altra apposita circuiteria per inviare questo *READY*?

Se il processore fosse collegato direttamente alla memoria, sì, nel caso in cui la memoria non sia in grado di rispondere nel tempo di ciclo stabilito dal processore.

Come viene utilizzato il registro *CR1*?

Il manuale del processore lo indica come *riservato*, quindi non si può usare. Vuol dire che l'Intel prevede di utilizzarlo in qualche modo in futuro (oppure che è stato introdotto per qualche motivo storico ormai dimenticato).

Nel libro si dice che l'assemblatore, durante la seconda passata, effettua i calcolo delle espressioni indirizzo che compaiono nei campi operandi delle varie istruzioni. Nel caso di istruzione operativa, in cui si utilizzi indirizzamento diretto o immediato, il valore dell'espressione calcolata, va a costituire il valore del campo *DISP* della codifica in linguaggio macchina. Nel caso di indirizzamento modificato, il valore dell'espressione indirizzo dipende anche dal valore contenuto in uno o più registri, non noto al momento dell'assemblaggio. Il valore immesso nel campo *DISP* è relativo soltanto alla componente spiazzamento dell'indirizzo simbolico?

Sì.

Il programma caricatore, nel caso sia necessaria la rilocazione, effettua la modifica di alcune parole lunghe contenenti indirizzi. Nel libro si dice che, per quanto riguarda la sezione *c_TEXT*, le parole lunghe che richiedono modifica, sono relative ad istruzioni operative che riferiscono variabili appartenenti alla sezione *c_DATA*, mentre, per la sezione *c_DATA*, sono quelle che contengono indirizzi della sezione *c_TEXT* o della sezione *c_DATA* stessa. Durante l'esecuzione del programma, il contenuto di una variabile, appartenente a *c_DATA*, può cambiare (può essere che la variabile sia inizializzata con un certo valore, che non rappresenta un indirizzo e che, in seguito, tramite una MOV, venga immesso in essa un indirizzo). Dato che la rilocazione avviene prima dell'esecuzione, la modifica del contenuto delle variabili della sezione *c_DATA*, è relativa solo al valore con cui esse vengono inizializzate? Viene modificato solo il contenuto di variabili inizializzate con espressioni indirizzo?

Sì

Nella sezione *c_TEXT*, possono esserci anche istruzioni operative che hanno come operando immediato un nome appartenente alla sezione *c_TEXT* stessa, del tipo:

```
movl $etichetta, destinazione
```

La destinazione, può poi essere usata come operando, ad esempio, per un'istruzione di salto (che non viene tradotta nel formato che prevede indirizzamento relativo rispetto a *EIP*, per il quale non c'è bisogno di rilocazione, ma nell'altro formato previsto per le istruzioni di controllo). In questo caso è necessario modificare il campo IMM della codifica dell'istruzione, sommandovi *D_c_TEXT*?

Sì, indipendentemente dall'uso che poi verrà fatto della variabile *destinazione* (uso che non può essere noto all'assemblatore).

Da quello che ho capito, senza tale modifica, nella destinazione, ad esempio una variabile della sezione *c_DATA*, oppure un registro, finirebbe un indirizzo errato. (il libro non prende in esame questo caso, ma parla solo di correzioni relative istruzioni operative che riferiscono operandi della sezione *c_DATA*).

Non ho qui con me il libro per verificare, ma mi pare che abbia ragione lei.

Capitolo 2

Memoria Cache

Perché il campo indice è stato posizionato nel mezzo e non al posto del campo etichetta?

Parliamo delle memorie ad indirizzamento diretto, che sono più semplici. Intanto, concentriamoci su un *blocco*, ovvero la sequenza di byte che viene gestita come se fosse un tutt'uno dalla cache. Si tratta di tutti i byte che hanno stesso indice e stessa etichetta. Possiamo, quindi, trascurare il campo spiazzamento e dire che ogni blocco ha un certo *indirizzo* $X = (E, I)$. Nelle memorie ad indirizzamento diretto, ogni blocco di memoria, di indirizzo $X = (E, I)$, ha una sua specifica posizione nella cache, quella data dal campo I. Però ci sono molti blocchi che hanno lo stesso indice e, quindi, *vorrebbero* andare nello stesso blocco della cache: tutti i blocchi di indirizzo (E', I), che hanno gli stessi bit nel campo indice, ma differiscono per l'etichetta. Al più uno alla volta di questi blocchi può trovarsi in cache (se questa è ad indirizzamento diretto) ad ogni istante, e questo è un problema per le prestazioni se un programma ha bisogno di due o più di questi blocchi *contemporaneamente* (cioè in tempi ravvicinati). Volendo restare nelle cache ad indirizzamento diretto, dobbiamo cercare di ridurre la probabilità che un programma possa aver bisogno contemporaneamente di due blocchi diversi con stesso indice. Ora, osserviamo come variano gli indici e le etichette dei blocchi in memoria, se l'etichetta è data dagli a bit più significativi dell'indirizzo del blocco, e l'indice è dato dai b bit seguenti: si parte dal blocco $(0, 0)$, quindi avremo $(0, 1), (0, 2)$ etc. fino a $(0, 2^{b-1})$. Quindi avremo $(1, 0), (1, 1), \ldots, (1, 2^{b-1})$ e così via. In pratica, vediamo che due indirizzi $X = (E, I)$ e $X' = (E', I')$ hanno lo stesso indice $I = I'$ solo se le etichette distano un multiplo di 2^b indirizzi. In altre parole, i blocchi che vorrebbero andare nella stessa posizione della cache sono abbastanza distanti in memoria. Poiché il principio di località ci dice che è molto probabile che un programma abbia bisogno *contemporaneamente* di blocchi vicini, più che di blocchi lontani, siamo a posto. Osserviamo invece cosa accade se il campo indice è dato dai b bit più significativi dell'indirizzo, e il campo etichetta dai restanti a bit. Ora, in memoria, avremo la sequenza di coppie (etichetta, indice) seguente: $(0, 0), (1, 0), (2, 0), ...(2^{a-1}, 0), (0, 1), (1, 1), (2, 1), ..., (2^{a-1}, 1)$. In questo modo, gli indirizzi che hanno stesso indice ed etichette diverse sono *consecutivi*, che è il peggio che potevamo fare.

Scrittura in cache (modalità write-back): quando c'è fallimento vado a scrivere in memoria centrale; se il fallimento è per il bit di validità ciò che ho scritto in memoria centrale si può copiare direttamente in cache nel blocco individuato dall'indirizzo (in memoria centrale e in memoria cache c'è lo stesso dato); se dipende dalle etichette diverse scrivo prima in memoria centrale. ...

Fin qui va bene.

... Dato che il blocco presente in cache è un'informazione valida c'è bisogno di un rimpiazzamento: quello scritto in centrale va in cache e quello della cache va nella stessa posizione in centrale dove avevamo scritto il nuovo blocco?

Qui no. Il blocco B_1 (valido e precedentemente modificato) che era in cache e va sostituito con il nuovo blocco B_2 andrà *al proprio posto* (di B_1) in memoria centrale, non al posto del nuovo (B_2). Il posto di B_1 è quello da cui B_1 era stato prelevato quando è stato portato in cache, e si trova semplicemente all'indirizzo (E_1, I_1), dove E_1 e I_1 sono rispettivamente l'etichetta e l'indice di B_1. Fare come dice lei non ha alcun senso. Inoltre, tenga presente che non è possibile che il posto proprio di B_1 coincida con quello di B_2, perché, altrimenti, le etichette sarebbero state uguali.

Nel caso di fallimento in scrittura con la write-through, dopo aver scritto in memoria centrale, si trasferisce il blocco in cache?

Nella cache descritta nel libro, no. Nel Pentium non accadeva, mentre nel Pentium Pro (e in quelli successivi) accade quello che dice lei: il blocco viene ricopiato in cache. Più in generale, esistono cache con politica write allocate (che trasferiscono il blocco in cache) e write no-allocate (che non lo fanno).

Con la politica write-back, nel caso di lettura che genera miss, bisogna trasferire anche il blocco sovrascritto in memoria centrale?

Ovviamente sì (se il blocco sovrascritto era stato modificato).

La memoria cache deve essere invalidata totalmente quando avviene una commutazione di contesto?

No.

Quando la cache viene invalidata e quando disabilitata?

Viene invalidata (= conteneva qualcosa e, dopo l'invalidazione, non la contiene più):

- quando il suo contenuto non corrisponde più al vero contenuto della memoria principale (per esempio perché il DMA ha modificato la memoria principale);

- quando la cache contiene informazioni più aggiornate rispetto a quelle contenute in memoria principale (ciò accade nelle cache di tipo write-back), vogliamo aggiornare la memoria principale con il contenuto della cache (per esempio perché deve essere letta da un dispositivo, come il DMA, che legge direttamente la memoria principale senza passare dalla cache) e il controllore cache non ci fornisce altro modo di farlo se non invalidando la cache (operazione che trasferisce in memoria principale le parti modificate presenti in cache).

In genere l'invalidazione non coinvolge l'intera cache, ma solo le parti che potrebbero contenere le zone di memoria principale interessate dai problemi precedenti (per esempio, solo gli indici corrispondenti agli indirizzi del buffer in memoria principale utilizzato dal DMA).

Viene disabilitata (= non contiene mai niente):

- quando non ha senso (dispositivi di I/O mappati in memoria);
- (meno fondamentale) quando non serve (programmi usati una sola volta, come il codice di bootstrap)

Anche qui, la disabilitazione non è totale (= per tutti gli indirizzi), ma coinvolge solo gli indirizzi di memoria principale interessati dagli ultimi due problemi precedenti.

In una cache ad indirizzamento diretto è possibile disabilitare, via software, un intervallo di gruppi, i cui estremi sono indicati dagli appositi registri. Se la cache è associativa ad insiemi, è possibile invalidare anche singoli gruppi all'interno di un insieme, oppure solo insiemi interi?

Non so se in qualche sistema ciò sia possibile. Mi sembra comunque poco utile: ciò che il programmatore vuole fare, in genere, è invalidare un certo intervallo di indirizzi della memoria centrale, nel senso che vorrebbe poter dire alla cache: se, per caso, stai memorizzando il contenuto di qualche byte all'interno di questo range, eliminalo (il programmatore potrebbe voler dire ciò perché sa che il contenuto in memoria a quegli indirizzi è cambiato rispetto a quello che potrebbe essere conservato in cache). Una interfaccia del genere, però, sarebbe abbastanza complessa da realizzare. Nelle cache ad indirizzamento diretto (o associative a insiemi) si offre normalmente una interfaccia più semplice, che è appunto quella descritta nel libro: si da solo la possibilità di invalidare un certo intervallo di gruppi nella cache. Siccome ogni range di indirizzi può trovarsi in cache solo in un certo intervallo di gruppi, invalidando quell'intervallo di gruppi siamo sicuri di rimuovere dalla cache ogni byte di quel range che, eventualmente, era conservato in cache. Ovviamente, usando questa semplice interfaccia, non sappiamo se veramente la cache memorizzava i byte che volevamo invalidare. Siamo però sicuri che, dopo aver impartito l'ordine di invalidazione, quei byte non si trovano in cache e una successiva lettura in quel range di indirizzi dovrà necessariamente andare a prelevare il contenuto in memoria. Tornando alla sua domanda, poichè i gruppi all'interno di un insieme sono del tutto equivalenti (vale a dire, un dato range di indirizzi di memoria centrale potrebbe essere conservato in uno qualunque dei gruppi all'interno dell'insieme), ha poco

senso dare la possibilità di invalidarli selettivamente (a meno che non si dia al programmatore anche la possiblità di sapere cosa i singoli gruppi contengono).

Nel libro si dice che la cache è usata per le pagine e non per le tabelle delle pagine (ragione per cui nei descrittori di tabella, i bit per il controllo della cache non sono significativi). Per le pagine virtuali occupate dalle tabelle delle pagine, la cache è abilitata o no? (mi riferisco allo stato del bit *PCD* nei corrispondenti descrittori).

Abilitata.

In fase si scrittura con la regola write-through, in caso di fallimento, come fa il processore a capire che deve scrivere nella memoria centrale e non nella cache?

Non lo capisce. La cache è completamente trasparente (vale a dire, il processore fa i suoi accessi in memoria ignorando la presenza della cache). È il controllore della cache che, in quel caso, scrive in memoria centrale.

Qual è il motivo per cui concettualmente la cache non funziona per lo spazio di I/O?

Usare la cache significa questo: invece di andare ogni volta in biblioteca a leggere le parti di un libro che mi interessano, fotocopio quelle parti e mi porto le fotocopie nell'ufficio. Nello spazio di I/O ci sono i registri delle interfacce, ad esempio il registro (di Input) in cui il controllore della tastiera scrive i codici dei tasti che premiamo, uno alla volta. Ha senso leggere questo registro una volta sola, copiarsene il contenuto in cache e poi rileggere sempre quello stesso valore dalla cache?

Perché nel montaggio si lascia comunque la possibilità di montare la cache nello spazio di I/O collegando il piedino *M/IO* al piedino */S*?

Attenzione, montare la cache nello spazio di I/O non c'entra assolutamente niente con il discorso precedente. La cache, oltre a svolgere la sua funzione, possiede anche dei registri (quelli che ci permettono di invalidarne il contenuto, tutto o in parte). Sono questi registri che vengono resi disponibili nello spazio di I/O o in memoria, a seconda di come montiamo la cache.

Quello che viene messo in cache durante la fase di reset viene invalidato oppure la cache *nasce* già con qualcosa di non casuale?

No, parte completamente *vuota*.

Il piedino */S* del controllore cache si attiva solamente quando si vuole accedere ai registri interni del controllore, oppure si attiva anche quando il processore cerca di fare un ciclo di lettura-scrittura in memoria? Se vale la prima, come si fa a disabilitare la cache durante i cicli di I/O? Mi è parso di capire che non è */DIS* che disabilita la cache in questo caso.

Fa un po' di confusione. La cache è, contemporaneamente, due cose:

- un dispositivo che intercetta tutti i cicli di bus del processore, prima che questi raggiungano il bus vero e proprio;
- una interfaccia con dei registri, come tutte le interfacce.

Disattivare la prima funzione significa far passare il ciclo di bus inalterato. Questo può avvenire su richiesta del processore (piedino */DIS*) oppure autonomamente da parte della cache stessa (poiché la cache intercetta anche il segnale *M/IO*, sa quando il ciclo di bus riguarda lo spazio di I/O e lo fa passare inalterato)

Disattivare, nel secondo caso, significa quello che significa per tutte le interfacce: l'interfaccia avrà i suoi registri a certi indirizzi dello spazio di I/O o dello spazio di memoria. Il piedino */S* deve essere attivato quando il processore esegue un ciclo di bus nello spazio (I/O o memoria) in cui si trova l'interfaccia, ad uno degli indirizzi a cui l'interfaccia ha i suoi registri, in modo che la richiesta di lettura o scrittura del processore venga servita da quell'interfaccia, con quel registro.

Nel metodo di scrittura write-back il blocco viene trasferito in memoria centrale solo in caso di rimpiazzamento. Ma vengono ricopiati in memoria centrale solo i blocchi con il bit V=1 oppure anche quelli con V=0?

Anatema! :) quelli con V=0 contengono informazioni non significative. Guai se fosse così.

Con riferimento alla figura di pagina 154 del libro sulle architetture, vorrei sapere se le maschere operano in questo modo: la maschera che manda l'uscita (attiva bassa) nella porta *AND* riconosce l'indirizzo (anzi, un range di indirizzi) della ROM (ovviamente in questo caso la cache deve essere disabilitata) ed eventualmente altri indirizzi per i quali vogliamo che la cache sia disabilitata (ad esempio il buffer utilizzato dal dma per i trasferimenti). L'altra maschera (quella che manda l'uscita nella porta *OR*) determina l'indirizzo a cui è montato il controllore, lo spazio è invece determinato dalla presenza o meno dell'inverter in basso. Il piedino CD è controllato dalla MMU, in base al bit *PCD* nel descrittore di pagina. Rimane un caso in cui dobbiamo disabilitare la cache: quando compiamo operazioni nello spazio di I/O. Il controllore cache riconosce da solo se l'operazione è di I/O (controllando il piedino *M/IO*) e si disabilita?

Sì, le cose stanno esattamente così.

Mi è sembrato di capire che se il controllore è selezionato è anche disabilitato e di conseguenza quando è abilitato non è selezionato! Deve essere così? Perché?

In realtà la figura parla di *maschere* (al plurale) e le due uscite (una che va alla porta *OR* e l'altra che va alla porta *AND*) non sono necessariamente la stessa uscita (anche perché, altrimenti, sarebbe stato disegnato un solo filo). Inoltre, la disabilitazione, pur se gestita dal controllore, è riferita alla cache, più

che al controllore: se la cache è disabilitata, gli accessi che il processore esegue sul bus la *attraversano*, come se la cache stessa non ci fosse. La selezione, invece, è per il controllore: il controllore è selezionato quando il processore vuole accedere ai registri interni al controllore stesso. In questo caso, dobbiamo pensare al controllore come a una normale interfaccia, cono dei registri, montata in memoria o nello spazio di IO. Come vede, le due cose (abilitazione e selezione) non hanno niente a che fare tra loro.

Capitolo 3

Interruzione di programma

Che cosa è il microprogramma di interruzione? Come è fatto e che ruoli svolge nell'ambito del meccanismo di interruzioni?

La CPU sarà costruita come una parte operativa e una parte controllo. In generale, la parte controllo sarà microprogrammata. Il *microprogramma di interruzione* è, appunto, il microprogramma eseguito dalla parte controllo ogni volta che si verifica una interruzione. Il suo ruolo è di svolgere tutte le operazioni associate alla ricezione dell'interruzione, fino al punto in cui la routine (o il task) corrispondente può partire quindi:

- ricezione del tipo;
- accesso alla corrispondente entrata della IDT;
- controllo della protezione;
- eventuale cambio di pila o commutazione hardware tra task;
- ...

Quando vengono accettate dal processore le eccezioni?

Nel momento in cui si generano, quindi, anche a metà dell'esecuzione di una istruzione.

Per le interruzioni esterne mascherabili il riconoscimento del tipo consiste in due letture nello spazio I/O ad indirizzi 0 e 4; il primo per notificare il controllore delle interruzioni che la richiesta è stata accettata e l'altro per prelevare il dato. A che serve leggere agli indirizzi 0 e 4 chi o cosa c'è montato nello spazio I/O? Per generare i due impulsi */INTA* che poi vanno al controllore di interruzione? ...

Sì

... Però il circuito di comando non ha in ingresso gli indirizzi e quindi per generare */INTA* non ha bisogno di sapere gli indirizzi 0 e 4.

Il fatto è che un qualche indirizzo deve essere comunque prodotto nei due cicli. Inoltre, i circuiti di pilotaggio del bus potrebbero essere diversi da quelli mostrati nel libro.

Le eccezioni sono asincrone?

No, tranne forse quelle della FPU.

Per quanto riguarda la tabella *IDT:* nel registro *IDTR* ci sta l'indirizzo base della tabella *IDT* presente sempre in memoria. L'indirizzo è fisico oppure virtuale? ...

Virtuale, nel senso che, se la paginazione è abilitata, verrà tradotto dalla MMU.

... E perché?

Nel processore Intel x86, tutti gli indirizzi sono virtuali, nel senso che tutti gli indirizzi passano dalla MMU prima di arrivare alla memoria fisica, è una scelta di progetto.

La differenza tra interruzione hardware e software è che la prima viene prodotta da circuiteria mentre l'altra dall'istruzione `INT` vero?

Sì (la circuiteria sarebbero le interfacce e il controllore delle interruzioni)

Che cosa fa esattamente un'eccezione di tipo trap?

Una eccezione (o interrupt) al cui tipo corrisponde un gate di tipo trap si comporta come nel meccanismo generale delle interruzioni:

- controllo del privilegio del gate (se si tratta di un interrupt software);
- confronto tra il livello di privilegio corrente e il livello di privilegio del segmento codice che contiene la routine di gestione, con eventuale cambio di pila;
- ...

L'unica differenza tra un gate di tipo interrupt e un gate di tipo trap è la gestione del flag *IF*, dopo il salvataggio in pila del vecchio valore di *EFLAGS*:

- i gate di tipo interrupt azzerano *IF;*
- i gate di tipo trap lasciano *IF inalterato*;

Quando va in esecuzione una routine relativa a un'interruzione che non è né di tipo trap, né di tipo interrupt, il controllo da parte del processore del bit TI rimane significativo? Il gate come viene classificato (se viene classificato)?

L'unica altra possibilità, se non è di trap/interrupt, è che sia di tipo task. In quel caso, il bit *TI* assumerà il valore che è stato salvato nel campo *EFLAGS* del segmento *TSS* del task entrante.

Come mai nel PC i controllori di interruzione sono entrambi programmati in modalità fully nested? non si otterrebbe lo stesso effetto programmando il master in fully nested e lo slave in modo normale?

Anche a me sembra di sì o, almeno, non mi viene in mente un controesempio per il momento. Le specifiche del chip *8259A*, però (http://bochs.sourceforge.net/techspec/intel-8259a-pic.pdf.gz) dicono che lo slave non può essere programmato in modo normale (modo *AEOI* = Automatic End Of Interrupt, nella terminologia Intel), senza fornire alcuna altra spiegazione.

Capitolo 4

DMA

Per trasferimento a ciclo unico potrebbe spiegarmi il problema del corto circuito in modo dettagliato?

Guardiamo la figura 11.2 a pagina 248 del libro bianco e teniamo presente che, non mostrati, a destra, ci sono i vari circuiti di abilitazione e comando del bus, quindi il bus con la memoria e almeno una interfaccia. Supponiamo di avere un trasferimento in ciclo unico da interfaccia a memoria. In questo caso, il controllore DMA inizia una normale operazione di scrittura in memoria, ma non pilota il bus dati. Il bus dati sarà invece pilotato dall'interfaccia. Però il transceiver non sa niente di tutto ciò (se non informato): vedendo una operazione di scrittura in memoria, abiliterà le sue porte tri-state verso il bus dati, creando un corto circuito con l'interfaccia che sta facendo altrettanto. Casomai vi venga in mente di dire: "*ma il transceiver ha in ingresso alta impedenza sul bus dati, quindi perchè la sua uscita dovrebbe fare corto cirtuito con l'uscita dell'interfaccia*", ripensateci bene. L'uscita di una porta non in alta impedenza è sempre 0 o 1, anche se in ingresso c'è alta impedenza (sarà 0 o 1 a caso, ma sempre 0 o 1 rimane).

Nel caso di trasferimenti con tipo di ciclo unico è necessario disabilitare i transceiver presenti sul bus, allo scopo di evitare problemi durante i trasferimenti verso la memoria. Da cosa derivano tali problemi? L'idea che ho io è la seguente: i trasferimenti in questione avvengono comandando un ciclo di scrittura in memoria, durante il quale, i piedini dati, *D31_D0*, del processore, fungono da piedini di uscita e sono connessi alle uscite del registro *MBR* (come visto a reti logiche). Questo può essere causa di conflitto al momento in cui l'interfaccia invia, sul bus, i dati da scrivere in memoria.

Non proprio: il problema è sulle uscite dei transceiver, non su quelle (che si trovano a monte) del processore. Nel ciclo unico, per esempio nel caso di trasferimento da interfaccia a memoria, il controllore DMA comanda un ciclo di scrittura in memoria tramite il bus e, contemporaneamente, ordina all'interfaccia (tramite un collegamento dedicato) di porre il dato sul bus dati. Il ciclo di scrittura in memoria passa, ovviamente, dai latch e transceiver e questi ultimi (se non disabilitati) abiliteranno le loro uscite nella direzione del bus dati, creando quindi un conflitto con l'interfaccia, che pilota lo stesso bus.

Durante il ciclo unico i transceiver presenti sul bus devono essere disabilitati. La condizione di corto si può generare sia per quel che riguarda il trasferimento da interfaccia a memoria che viceversa?

No, nel caso da memoria a interfaccia non ci sono problemi (purché, ovviamente, il controllore DMA ponga in alta impedenza le proprie linee dati in uscita).

Non ho capito perché in condizioni di controllore dma slave, c'è pericolo di corto solo quando il processore legge dal controllore, ma non quando vi scrive. Non c'è in entrambi i casi il pericolo che qualcosa possa arrivare dai transceiver?

No, se il processore esegue una scrittura (di qualunque tipo), i transceiver saranno abilitati verso il bus, non verso il processore.

Nelle slide viste a lezione, il pin */READY* del DMA era solo in ingresso. Il mio dubbio è: nel colloquio tra DMA e processore, il segnale di */READY* dovrebbe essere attivato da qualcuno, e se non è il DMA, di chi si tratta?

Il segnale */READY* serve ad inserire cicli di attesa nelle operazioni sul bus, quando l'operazione non può concludersi nel tempo stabilito a-priori, per qualche motivo. Potrebbe essere usato, ad esempio, da qualche periferica, più lenta delle altre, per allungare la durata dei cicli di lettura o scrittura che la coinvolgono. È chiaro, allora, che anche il DMA ha bisogno di ricevere in ingresso tale segnale, nel caso dovesse essere programmato per leggere o scrivere da tale periferica.

Il problema del cortocircuito nel caso del controllore DMA e avviene solo nel caso di ciclo unico? Che differenza c'è con il ciclo doppio?

Nel ciclo unico (ad esempio in scrittura), il controllore DMA comanda una operazione di scrittura in memoria, ma non fornisce i dati sul bus: lascia che sia direttamente l'interfaccia a fornirli. Il transceiver, però, nel vedere una operazione di scrittura in memoria, abiliterà (se non preventivamente disabilitato) le sue porte che pilotano le linee D sul bus dati (perché, normalmente, i dati da scrivere in memoria arrivano dal processore che si trova prima del transceiver) creando un cortocircuito con le porte dell'interfaccia. Nel ciclo unico in lettura, il problema si presenta tra il transceiver e le porte del controllore DMA che pilotano le linee dati, ma, in questo caso, il corto circuito si può evitare anche facendo in modo che il controllore DMA ponga tali porte in alta impedenza. Nel ciclo doppio tutti questi problemi non si pongono, in quanto i controllore DMA esegue due operazioni di lettura/scrittura distinte, comportandosi esattamente come si comporterebbe il processore.

Quando il controllore DMA diventa master ha il possesso del bus e quindi effettua autonomamente cicli di bus. Il processore quindi durante l'operazione di trasferimento in DMA non può effettuare cicli di bus. Mi chiedo quindi come sia possibile leggere il contenuto di *TCR* per realizzare ad esempio una percentuale di avanzamento dell'operazione o semplicemente per leggere il valore del registro di stato *STR* ad esempio per un'eventuale gestione a controllo di programma della terminazione.

Se il modo di trasferimento è singolo (cycle stealing), il controllore DMA richiede e rilascia il bus per ogni trasferimento elementare (cioè per ogni byte, se l'interfaccia è a 8 bit). Il processore ha il tempo di fare quello che lei dice tra un trasferimento e il successivo. Se il modo di trasferimento è a ciclo continuo (burst), il bus è occupato per tutto il trasferimento e il processore non può effettivamente leggere *TCR* o STR. Volevo però farle notare che stiamo parlando di tempi microscopici, non macroscopici. Ad esempio, il DMA verrà programmato per leggere un blocco (1000/8000 byte) di un file in memoria, operazione che richiede pochi millisecondi nel peggiore dei casi. Quindi non avrebbe proprio il tempo di vedere la percentuale di avanzamento. Per trasferire un intero file, il controllore DMA verrà riprogrammato molte volte, e la percentuale di avanzamento può essere aggiornata ogni volta che termina il trasferimento di un blocco.

Perché nella `dmaleggi`() viene fatta prima la `dmastart_in`() e poi la `sem_wait`() sul semaforo di sincronizzazione? E perché nel processo esterno di fine lettura DMA viene effettuata la `sem_signal`() su tale semaforo?

Le due cose sono collegate. Il processo che ha invocato la `dmaleggi`() vuole *aspettare* che la lettura appena richiesta sia terminata. Il processo esterno associato all'operazione di dma su questa interfaccia *sa* quando l'operazione è terminata (infatti, questo è proprio l'evento che ha messo in esecuzione il processo). Il processo che vuole aspettare, chiamiamolo P, esegue una `sem_wait`() sul semaforo di sincronizzazione (cioè inizializzato a 0). Siccome il semaforo era a 0, il processo verrà sospeso e messo nella coda del semaforo (liberando così il processore, che può utilmente eseguire altri processi). Quando il processo esterno esegue la `sem_signal`(), risveglierà il processo P (notare che il semaforo resta comunque a 0, pronto per il prossimo utilizzo). Il meccanismo funziona lo stesso anche se, per caso, il processo riesce ad eseguire la `sem_signal`() prima che P riesca ad eseguire la `sem_wait`(): in questo caso il semaforo passa prima a 1 (con la `sem_signal`()) e poi torna a 0 quando P esegue la `sem_wait`(). In questo sceanario P non si blocca.

Non mi è chiaro il problema del cortocircuito sul bus dati. Perché nel tranceiver viene usato il piedino */OE* mentre nel Latch no? Quali sono i casi in cui si verifica il corto circuito?

Quando due porte diverse pilotano attivamente (quindi, con i livelli 0/1 e non con il terzo stato) lo stesso collegamento. Il transceiver può pilotare anche i collegamenti dati verso il processore, mentre il latch no (solo verso il bus). Il problema si presenta quando il processore deve dialogare con il controllore DMA,

che si trova prima del transceiver: per leggere un registro del controllore DMA, il processore eseguirà una operazione di lettura nello spazio di memoria (dove il controllore è tipicamente montato), a cui il controllore risponderà pilotando le linee dati (con il contenuto del registro richiesto). Ma anche il transceiver (se non disabilitato), vedendo una operazione di lettura (che, normalmente, prevede che i dati arrivino dal bus che si trova oltre il transceiver stesso), piloterà le stesse linee dati, creando il cortocircuito.

Capitolo 5

Cache e DMA

Nel libro si dice che è possibile far sì che i piedini di indirizzo del controllore cache, quando è attivo il segnale *HOLDA*, fungano anche da piedini di ingresso, cosicché gli indirizzi generati dal controllore DMA producano un accesso ad un blocco della memoria cache e, nel caso di successo, la sua invalidazione, allo scopo di mantenere consistente il contenuto della memoria cache con quello della memoria centrale, anche nel caso di trasferimenti in DMA. L'invalidazione si ha anche nel caso in cui si esegua in DMA un trasferimento da memoria ad interfaccia? (in questo caso i dati in memoria, coinvolti nel trasferimento non vengono modificati)

In questo caso si ha il problema inverso: il contenuto della memoria potrebbe essere più vecchio rispetto a quello contenuto nella cache. Questo caso è più complesso, in quanto l'interfaccia deve ricevere il dato contenuto in cache e non quello contenuto in memoria. Una possibile soluzione prevede che il controllore cache, una volta accertato che l'indirizzo richiesto sul bus è presente in cache, completi il ciclo di lettura al posto della memoria (è necessario però qualche accorgimento per fare in modo che la memoria non cerchi di rispondere contemporaneamente, creando un conflitto sul bus dati).

Una volta terminato il trasferimento, supponiamo un'operazione di ingresso, per accedere ai nuovi dati, il programma, non può usare la cache?

Nel caso in questione, no (la cache è permanentemente disabilitata).

Nel caso di trasferimenti DMA tra memoria di massa e memoria centrale, con cache abilitata (col metodo write-back), come fa il controllore cache a sapere se viene fatto un accesso in scrittura alla memoria centrale (con invalidazione del blocco) oppure in lettura?

Potrebbe invalidare in ogni caso, per sicurezza. Comunque, se vogliamo farlo più preciso, ci basta mettere in ingresso tutte le linee del bus, comprese le linee di controllo che ci dicono se, appunto, il ciclo è di lettura o di scrittura (i PC IBM compatibili sono fatti così).

Durante i cicli di lettura in memoria da parte del DMA, il dispositivo accede prima alla memoria cache? In caso negativo può spiegarmi il motivo?

No, non accede alla memoria cache. Non può farlo, perché il controllore DMA è collegato direttamente al BUS, senza passare dal controllore cache. Comunque, il controllore cache può essere abbastanza *furbo* da ascoltare cosa accade sul BUS e capire come comportarsi in modo da garantire la correttezza degli accessi in memoria anche in presenza di DMA.

Quando si collega il DMA alla cache perché solo *HOLD* del processore è messo a terra? Non dovrebbero essere messi a terra tutti e due?

E perché mai? *HOLD* serve al controllore DMA per chiedere l'accesso esclusivo al BUS. Poiché, in presenza del controllore cache non è il processore che accede al bus, ma il controllore cache stesso, il controllore DMA deve chiedere l'accesso a quest'ultimo e non al processore (collegamento tra *HOLD* in uscita dal controllore DMA a *HOLD* in entrata nel controllore cache). Poiché, a questo punto, nessuno chiederà mai al processore l'accesso al bus, mettiamo il suo ingresso HOLD costantemente a 0.

Quando il DMA genera indirizzi il controllore Cache controlla se riesce ad effettuare l'accesso in cache e in caso di successo invalida quelle informazioni. È corretto?

È necessario invalidare solo quando il DMA scrive in memoria, non quando legge. Chiaramente, per fare questa distinzione il controllore cache deve avere in ingresso anche W/R.

Se il controllore DMA deve leggere in memoria centrale per inviare dati all'interfaccia potrebbe leggere in cache per avere una maggiore velocità? Se si come fa a dare il comando di lettura al controllore cache se *W/R* non è di ingresso a quest'ultimo?

La cosa è più complessa. Se la cache è write-back il controllore cache deve fare qualcosa per forza, altrimenti il DMA potrebbe leggere dati vecchi dalla memoria (in altre parole, non si tratta soltanto di una questione di prestazioni). Quello che può fare è scrivere in memoria le linee modificate coinvolte nell'operazione di DMA (e poi lasciare che il DMA legga dalla memoria), oppure rispondere lui stesso all'operazione di lettura. Anche qui, serve che *W/R* sia in ingresso (tra l'altro). Se *W/R* non è in ingresso, otteniamo un controllore cache molto più semplice, che si limita a invalidare (e quindi scrivere in memoria, se modificate) tutte le linee coinvolte in una operazione di DMA, sia essa di lettura o di scrittura.

Pagina 253, libro bianco: *nella realizzazione delle operazioni di I/O, i trasferimenti [...] in tal modo una modifica del contenuto del buffer non ha alcuna ripercussione sulla cache stessa*. Non mi è molto chiaro. La cache per le operazioni di I/O è disabilitata, quindi perché usare il buffer?

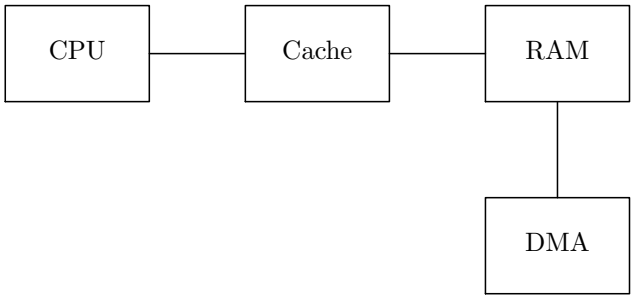
Perché il controllore DMA legge i dati da inviare all'interfaccia dalla memoria (o viceversa, li legge dall'interfaccia e li scrive in memoria).

Mi pare di aver capito che il buffer usato per i trasferimenti in DMA da un certo programma, occupa indirizzi fisici per i quali la cache è disabilitata (dalla maschera la cui uscita va in ingresso alla porta *AND* collegata al piedino */DIS*), che gli vengono assegnati da un'apposita routine di nucleo (quella che nel libro è chiamata *trasforma*).

No, la routine *trasforma* si limita a trasformare da virtuale (lineare) a fisico l'indirizzo passatole come argomento. Il programma che usa il DMA deve già sapere (a-priori) che, per quell'indirizzo virtuale, la cache è disabilitata (per esempio, il sistema potrebbe predisporre un range di indirizzi destinati ai trasferimenti via DMA).

Perché il buffer delle operazioni DMA non deve essere soggetto al meccanismo della memoria cache?

Perché il controllore DMA, che deve leggere o scrivere in quel buffer, bypassa la cache. Lo schema è il seguente:



Quindi il controllore DMA non può sapere cosa è in cache. Ciò causa due problemi:

- nel caso di trasferimento da RAM ad I/O, il controllore potrebbe leggere in RAM dati *vecchi* rispetto a quelli contenuti in cache (se questa usa il meccanismo write-back);
- nel caso di trasferimento da I/O a RAM, il processore potrebbe leggere (in cache) dati *vecchi* rispetto a quelli contenuti in RAM (cioè quelli appena scritti dal controllore)

Parte II

Aspetti architetturali avanzati e nucleo di sistema operativo

Capitolo 6

Memoria Virtuale

Non riesco a scrivere in Assembler cosa fa la MMU quando, in seguito alla generazione da parte di un programma di un indirizzo di memoria, provvede ad effettuare la traduzione di indirizzo da virtuale a fisico.

Qualcosa del genere:

```
movl  %cr2, %eax             # indirizzo virtuale non tradotto in %eax
movl  %eax, %ebx             # lo copiamo in %ebx (ci servirà più volte)
movl  %ebx, %ecx             # registro di lavoro
andl  $0xffc00000, %ecx      # isoliamo i 10 bit più significativi
shl   $22, %ecx              # e otteniamo l'indice nel direttorio
movl  %cr3, %eax             # base del direttorio in %eax
movl  (%eax, %ecx, 4), %eax  # descrittore di tabella in %eax
testb $1, %al                # bit di presenza
jz    solleva_page_fault     # se 0 manca la tabella delle pagine
movb  $0, %al                # azzeriamo il byte di accesso
                             # (ora %eax contiene l'indirizzo della tabella)
movl  %ebx, %ecx             # riprendiamo l'indirizzo virtuale non tradotto
andl  $0x003ff000, %ecx      # isoliamo i 10 bit centrali
shl   $12, %ecx              # e otteniamo l'indice nella tabella delle pagine
movl  (%eax, %ecx, 4), %eax  # descrittore di pagina in %eax
testl $1, %al                # bit di presenza
jz    solleva_page_fault     # se 0 manca la pagina
movb  $0, %al                # azzeriamo il byte di accesso
                             # ora %eax contiene l'indirizzo fisico di pagina
movl  %ebx, %ecx             # ancora l'indirizzo virtuale non tradotto
andl  $0x00000fff, %ecx      # isoliamo l'indirizzo di riga
orl   %ecx, %eax             # ora %eax contiene l'indirizzo tradotto
```

Il campo indirizzo dei descrittori di tabelle presenti nel direttorio cosa contiene?

Quando la tabella puntata è presente in memoria fisica, contiene i 20 bit più significativi dell'indirizzo fisico della tabella (i rimanenti 12 bit devono essere 0, in quanto l'indirizzo della tabella deve essere allineato alla pagina). Quando la tabella puntata non è presente in memoria fisica, facciamo l'ipotesi che tutto il

descrittore (e quindi anche il campo indirizzo) contenga l'indirizzo della tabella in memoria di massa (cioè, tutte le informazioni necessarie per reperire la tabella in memoria di massa).

Se il campo indirizzo non viene usato per contenere le informazioni necessarie per reperire la tabella in memoria di massa come si trova alternativamente la pagina richiesta?

È necessaria un'altra struttura dati. Ad esempio, nei sistemi con file system (quasi tutti), il file che contiene un eseguibile contiene anche le informazioni necessarie al mappaggio del file in memoria.

Pagina 18, volume IV: "*Consideriamo la situazione iniziale in cui sono presenti in memoria fisica solo il direttorio e le tabelle delle pagine che consentono di posizionare il direttorio, quelle tabelle delle pagine e la memoria fisica in memoria virtuale*". Io ho capito che in memoria virtuale sta la memoria fisica, che a sua volta contiene il direttorio e le tabelle delle pagine che consentono di posizionarlo. Vorrei sapere se ho capito bene o se mi è sfuggito qualcosa.

Ha capito bene, nel senso che mappare tutta la memoria fisica in memoria virtuale è già sufficiente per fare tutto ciò che serve (compreso, come dice lei, poter accedere al direttorio e alle tabelle). In quella versione del libro, però, il professore pensava ad un meccanismo aggiuntivo per posizionare direttorio e tabelle in memoria virtuale, mentre la mappatura della memoria fisica in memoria virtuale doveva avere il solo scopo di permettere alla routine di page fault di poter leggere/scrivere una qualunque pagina fisica. Presto dovrebbe uscire una nuova versione del libro, in cui il primo meccanismo è stato eliminato, visto che è superfluo.

La routine di trasferimento come fa a modificare il direttorio se quest'ultimo non è in memoria virtuale e lei conosce solo il suo indirizzo fisico? O meglio, come si mappa il direttorio in memoria virtuale?

Il direttorio deve essere in memoria virtuale per poter essere modificato via software dalla routine di trasferimento. Ovviamente, la routine di trasferimento deve conoscerne l'indirizzo virtuale. Mappare qualunque cosa in memoria virtuale significa fare in modo che una qualche tabella delle pagine vi punti. Il direttorio non fa eccezione. Supponiamo che il direttorio si trovi all'indirizzo fisico F (cioè, che F sia il contenuto di *CR3*) e che vogliamo mapparlo all'indirizzo virtuale V, scomposto in $V = (v_1, v_2, v_3)$ (dove v_1 corrisponde ai 10 bit più significativi di V, v_2 sono i 10 bit seguenti e v_3 i 12 bit rimanenti). Dovremo avere una tabella delle pagine già in memoria fisica, poniamo all'indirizzo fisico F', per fare in modo che il descrittore v_1-esimo (contando da 0) del direttorio punti a F', e che il descrittore v_2-esimo (contando da 0) della tabella punti a F. A questo punto, scrivendo o leggendo agli indirizzi virtuali $[V, V + 4096)$, scriviamo e leggiamo agli indirizzi fisici $[F, F + 4096)$, cioè nel direttorio. Tutto ciò possiamo predisporlo prima di abilitare la paginazione, in modo da poter accedere liberamente a tutta la memoria fisica e preparare il direttorio e la tabella come meglio ci piace. Tutto ciò vale come discorso generale perché, se

abbiamo già mappato la memoria fisica in memoria virtuale, non dobbiamo fare nient'altro: l'indirizzo fisico F, contenuto in *CR3*, sarà già mappato in memoria virtuale, all'indirizzo virtuale $V = F$.

Quando viene generata un'eccezione page-fault (nel descrittore di tabella all'interno del direttorio abbiamo P=0) e va in esecuzione la routine di trasferimento, dopo aver trasferito la tabella delle pagine opportuna in memoria fisica, come fa la routine a scrivere l'indirizzo della tabella delle pagine nel descrittore di tabella del direttorio? Gli è sufficiente l'indirizzo virtuale memorizzato in *CR2* che ha generato il fault?

L'indirizzo virtuale memorizzato in *CR2* permette alla routine di sapere quale entrata del direttorio deve essere modificata (usa i 10 bit più significativi di *CR2* come indice nel direttorio). In quella entrata, la routine dovrà scrivere l'indirizzo fisico della tabella.

Se il direttorio *sta* in memoria virtuale vuol dire che ci deve essere un *altro* indirizzo virtuale che punta a una tabella di pagine, all'interno della quale vi è un descrittore di pagina che punta al direttorio?

Esatto (non solo per il direttorio, ma per qualunque cosa si trovi in memoria virtuale).

Questo indirizzo virtuale si deve stabilire a priori? E la routine come fa a conoscerlo?

È il sistema stesso che decide l'indirizzo virtuale: ci sarà una routine di inizializzazione che, prima di attivare la paginazione, predispone la tabella che mappa il direttorio in memoria virtuale. Quindi, è sufficiente che la routine di inizializzazione lasci questo indirizzo in una variabile globale, accessibile alla routine di page fault. Se, però, si mappa tutta la memoria fisica in memoria virtuale, la routine di page fault può usare direttamente l'indirizzo fisico contenuto in *CR3*, come se fosse virtuale.

La routine non *vede* memoria virtuale?

Certo, come tutte le routine software, una volta che è stata abilitata la paginazione.

Siccome la MMU opera in modo automatico non bisognerebbe che la routine emanasse quell'indirizzo virtuale che una volta tradotto va a individuarmi direttamente il descrittore da modificare?

Certo.

E allora come fa la stessa routine a usare i 10 bit più significativi di *CR2* insieme al contenuto di *CR3* per formare quell'indirizzo virtuale che una volta tradotto mi permette di accedere al descrittore di tabella dove bisogna scrivere?

L'uso diretto di *CR3* è possibile se provvediamo prima a mappare tutta la memoria fisica in memoria virtuale a partire dall'indirizzo virtuale 0, come suggerito nel testo. Se siamo in questa situazione (*praticamente* fattibile, se la memoria fisica installata è meno di 1GiB), possiamo accedere a qualunque indirizzo fisico utilizzandolo come se fosse virtuale.

La tabella delle pagine che mi mappa il direttorio in memoria virtuale è fatta in modo che ogni suo descrittore contenga l'indirizzo fisico di un descrittore del direttorio (il primo descrittore della tabella contiene l'indirizzo fisico del primo descrittore del direttorio)?

No. Il direttorio è grande 4KiB, quindi è sufficiente un'unica entrata della tabella per mapparlo interamente in memoria virtuale.

Se la routine per scrivere nel direttorio conosce l'indirizzo virtuale del direttorio V che è tradotto da quello fisico F, per scrivere nel descrittore di tabella apposito, emette l'indirizzo virtuale dato dalla somma tra l'indirizzo virtuale V del direttorio e i dieci bit più significativi di *CR2*? Lo stesso per scrivere in un descrittore di pagina somma all'indirizzo virtuale V_1 della tabella delle pagine (mappata in memoria virtuale) i secondi dieci bit di *CR2*?

Sì, ma il valore dei 10 bit va prima moltiplicato per 4, in entrambi i casi, perché ogni descrittore (sia di tabella che di pagina) è grande 4 byte.

Gentile prof, lei ha detto che è fattibile mappare tutta la memoria fisica in memoria virtuale se tale memoria fisica è al massimo 1 GiB. Perché?

Non ho mai detto questo. Ho detto che se si mappa tutta la memoria fisica in memoria virtuale, si riduce ovviamente lo spazio disponibile per la memoria virtuale vera e propria. Chiaramente, se la memoria fisica installata fosse di 4 GiB (non 1) e la mappassimo tutta in memoria virtuale, lo spazio per la vera memoria virtuale sarebbe 0, quindi la soluzione sarebbe impossibile. Per valori minori di 4 GiB si può sempre fare (almeno in teoria), ma è meglio che siano molto minori. Per esempio, vecchie versioni di Linux permettevano di mappare tutta la memoria fisica in memoria virtuale fino a dimensioni della memoria fisica di 3 GiB (lasciando quindi almeno un 1 GiB per la vera memoria virtuale).

Quando la memoria fisica va in memoria virtuale perché deve essere messa nella posizione iniziale? Solo per una questione di coincidenza tra indirizzo virtuale-fisico in modo che non ci sia bisogno di traduzione?

Non *deve*, è comodo averla in quella posizione in modo che, nel momento in cui si abilita la memoria virtuale, ci sia continuità nell'indirizzamento. La *traduzione*

c'è in ogni caso, nel senso che la MMU c'è sempre e continua a fare il suo mestiere, solo che, per quel range di indirizzi, ottiene un indirizzo fisico uguale a quello virtuale.

La routine di trasferimento per andare in memoria di massa deve recuperare l'informazione nel descrittore di pagina che ha il *P*=0 e che ha generato il page-fault. Per farlo, dato che è una routine, non può accedere a *CR3* come fa la MMU (perché esso contiene un indirizzo fisico) e quindi deve andare per forza in memoria. Come fa a sapere dove inizia il direttorio in modo che con i primi 10 bit dell'indirizzo virtuale di pagina individui il corrispondente descrittore di tabella? Inoltre, anche se riuscisse a far questo, incontrerebbe un indirizzo fisico di tabella delle pagine, che non può usare. E allora come fa a sapere dove si trova il descrittore di pagina, o di tabella, dal quale preleva l'indice per andare in memoria di massa?

Questo è uno dei motivi per cui mappiamo la memoria fisica in memoria virtuale. Se la mappiamo a partire dall'indirizzo virtuale 0, avremo virtuale=fisico per tutti gli indirizzi virtuali minori della dimensione della memoria fisica installata, quindi la routine di page-fault può usare direttamente il contenuto di *CR3* come indirizzo del primo byte del direttorio, e lo stesso vale per gli indirizzi che trova nei descrittori di tabella.

Quando occorre effettuare un trasferimento di tabella delle pagine, quale indice (non so se in effetti si può chiamare indice il gruppo di bit che trova nel descrittore e che nel caso *P*=1 è l'indirizzo fisico) si considera per andare in memoria di massa? Quello del direttorio o quello della tabella che la fa risiedere in memoria virtuale?

Quello del direttorio. Comunque, il fatto di usare il descrittore di tabella/pagina anche quando la tabella/pagina non è presente (per memorizzarvi l'indirizzo in memoria di massa) è una cosa realizzata in software. In altre parole, non ha senso rispondere alla sua domanda se non facendo riferimento ad un particolare sistema operativo. La risposta che le ho dato (quello del direttorio) è solo la più comune e più naturale.

Se è quello del direttorio le modifiche vanno apportate anche nel descrittore di pagina della tabella che la fa risiedere in memoria virtuale?

Anche qui, la risposta dipende dal modo in cui è realizzata la mappatura della tabella di corrispondenza in memoria virtuale. Se è realizzata tramite la mappatura della memoria fisica in memoria virtuale, non c'è bisogno di fare quello che dice lei. Altrimenti, diciamo di sì.

Pagina 20, volume IV: "*La MMU, nel tradurre un indirizzo da virtuale a fisico, prima accede al TLB e, solo in caso di fallimento, al direttorio e alla tabella delle pagine*". Pagina 21, volume IV: "*Il TLB produce successo se [...]*". Ma il TLB non è semplicemente un buffer? Non è quindi la MMU a generare successo o fallimento a seconda delle condizioni?

Nell'implementazione riportata nel libro, una parte del segnale di successo o fallimento è sicuramente generato dal TLB (da una serie di comparatori per le etichette, in *AND* con i rispettivi bit di validità). Le altre condizioni (sui bit *D*, *S/U* e *W/R*) sono più complesse e possono essere attribuite alla MMU o al TLB. In ogni caso, non è una distinzione così importante o significativa.

Pagina 21, volume IV: "*Quando avviene un accesso in scrittura a una pagina in cui il descrittore è presente nel TLB, ma ha il bit D uguale a zero, si ha un fallimento, e la MMU effettua un accesso in memoria al direttorio e alla tabella delle pagine coinvolta: effettua la traduzione di indirizzo, il descrittore di pagina viene quindi modificato (il bit D passa da 0 a 1) e di nuovo trasferito nella stessa posizione del TLB*". Chi è che modifica il bit *D* del descrittore di pagina? Perché non si modifica direttamente *D* nel TLB?

La MMU modifica il bit *D* del descrittore di pagina. Non è possibile modificarlo direttamente nel TLB perché la routine di trasferimento non può *leggere* il bit *D* nel TLB.

Continuando a leggere trovo un accenno alla routine di trasferimento e quindi, affinché essa possa essere mandata in esecuzione, deduco che debba essersi generata l'eccezione di page-fault. In quale preciso momento viene generata questa eccezione?

L'accenno alla routine di trasferimento è legato al fatto che si sta parlando del bit *D*, il cui valore è letto da tale routine. La routine andrà in esecuzione quando si verificherà un page fault, non va pensata in esecuzione mentre avviene il fallimento sul TLB. Il fallimento del TLB non c'entra niente con il page fault.

Supponiamo che la tabella di corrispondenza venga fatta risiedere in memoria virtuale in modo tale che sia possibile modificarla via software senza accedere alla parte di memoria virtuale riservata alla memoria fisica, se non sbaglio questo viene fatto in modo tale che la routine di timer possa facilmente accedere via software ai vari descrittori resettando il bit *A* e in modo tale che anche la routine di rimpiazzamento possa accedere ai descrittori per modificarli nel caso di trasferimento di pagine. Ogni volta che si traduce un indirizzo da virtuale a fisico la MMU accede prima al direttorio e (se non lo è già) mette ad 1 il bit *A* nel descrittore di tabella del direttorio (perché acceduta) e ad 1 anche il bit *A* nel descrittore di pagina nella tabella delle pagine selezionata, in particolare, se via software tento di accedere alla tabella delle pagine che è in memoria virtuale, dovrei mettere ad 1 il bit *A* nel descrittore di pagina che la fa risiedere in memoria virtuale, giusto?

Giusto (se intendiamo quel "dovrei mettere ad 1 il ..." come "la MMU mette ad 1...").

Però così facendo avrei un descrittore di tabella che contiene l'indirizzo fisico della tabella, e un descrittore di pagina (contenuto in una tabella delle pagine) che contiene lo stesso indirizzo, tuttavia il bit *A* nel direttorio viene aggiornato ogni volta che accedo alla tabella e dunque è effettivamente corretto per le politiche di rimpiazzamento, perché necessariamente $\geq$ a tutti i bit *A* nei descrittori di pagine all'interno della tabella. Il bit *A* interno al descrittore di pagina nella tabella delle pagine che riferisce la tabella delle pagine considerata però viene incrementato solo quando vi accedo generando gli appositi indirizzi virtuali. Questo bit *A* ha un significato per quanto riguarda il rimpiazzamento? Come fa dunque la routine di timer a capire quali bit *A* sono associati a pagine che sono in realtà tabelle delle pagine? e come li tratta?

Come ho scritto molte volte, domande di questo tipo sono mal poste. La routine di timer è una routine software, quindi non possiamo sapere *cosa fa* se non facendo riferimento ad una particolare implementazione. Possiamo chiederci cosa *può* o *deve* fare una qualunque routine di timer soggetta agli stessi vincoli ma, trattandosi di software, molte di queste domande non hanno una risposta univoca o precisa. Nel caso della sua domanda, la routine *può* avere una struttura dati a parte che le dice il tipo di contenuto delle varie parti dello spazio di indirizzamento, oppure *può* usare i bit disponibili nei descrittori di pagina/tabella per discriminare pagine soggette a rimpiazzamento da pagine usate per mappare la tabella i corrispondenza, oppure qualunque altro metodo che può venire in mente al programmatore.

Quando si verifica un page-fault viene salvato in *CR2* l'indirizzo virtuale che ha generato fault e in pila il contenuto di *CS* e *EIP* relativi all'istruzione da rieseguire. La routine che va in esecuzione chiama la routine vera e propria (`CALL _c_page_fault`). Poiché il trasferimento di una pagina in memoria fisica richiede un po' di tempo, questa routine viene eseguita con le interruzioni abilitate, utilizzando però un semaforo relativo al corpo rientrante della routine in modo che, nel caso vada in esecuzione un altro processo e questo provochi page-fault, non sia possibile richiamare la routine finché non abbia terminato il trasferimento. A questo punto viene preso l'indirizzo fisico del direttorio dal registro *CR3* e, poiché abbiamo mappato la memoria fisica in memoria virtuale a partire dall'indirizzo 0, avremo corrispondenza fra indirizzo fisico e indirizzo virtuale. In questo modo, attraverso i 10 bit più significativi dell'indirizzo virtuale che ha generato fault, accediamo al descrittore di tabella e verifichiamo se il suo bit *P*=0. In tal caso significa che la tabella non è presente in memoria fisica, quindi prendiamo il contenuto del campo indirizzo del descrittore di tabella che servirà come coordinate per recuperare tale tabella in memoria di massa. Si esegue il rimpiazzamento attraverso l'apposita routine (che in caso di memoria fisica piena selezionerà una pagina da sostituire attraverso le statistiche fatte dalla routine di timer) e si aggiornerà il contenuto del descrittore di tabella inserendo nel campo indirizzo l'indirizzo fisico della pagina fisica in cui è stata caricata. A questo punto (sia che la tabella fosse presente, sia che non lo fosse) si considera l'indirizzo fisico della tabella e vi si accede (sempre per il fatto di avere la memoria fisica mappata nella memoria virtuale) selezionando l'apposito descrittore di pagina attraverso i bit da 21 a 12 dell'indirizzo virtuale che ha generato fault. Poiché la pagina sarà sicuramente non presente in memoria fisica...

Non è detto. Nel codice del nucleo ci dovrebbe essere un commento che spiega in quali casi la pagina potrebbe, invece, essere già presente. Si tratta comunque di un dettaglio secondario.

. . . si provvede a recuperarla in memoria di massa attraverso le coordinate contenute nel campo indirizzo del descrittore di pagina, si richiama la routine di rimpiazzamento e si aggiorna il contenuto del descrittore di pagina con l'indirizzo fisico della pagina appena caricata. A questo punto la routine termina e attraverso la sua `IRET` vengono recuperati dalla pila i contenuti di *CS* ed *EIP* relativi all'istruzione che ha generato fault in modo che questa possa essere rieseguita.

Pagina 14, volume IV: "*la memoria fisica, per poter essere letta o scritta via software, deve trovarsi anche in memoria virtuale*". Se il direttorio e le tabelle delle pagine stanno in memoria fisica, allora, sotto l'ipotesi precedente sono anche in memoria virtuale. Perché si rende necessario porre anche la tabella di corrispondenza (direttorio e tabelle delle pagine) in memoria virtuale? Direttorio e tabelle delle pagine non sarebbero già accessibili (almeno le tabelle delle pagine presenti in memoria fisica) dal software?

No, non è necessario. Come lei fa giustamente notare, mappare la memoria fisica in memoria virtuale permette già di accedere al direttorio e a tutte le tabelle delle pagine presenti. Le due tecniche illustrate nel libro sono più alternative che complementari.

Che significa che una certa entità è residente in memoria virtuale?

Fuori dal contesto, la frase è un po' vaga. Tutte le entità accessibili al programmatore sono residenti in memoria virtuale, perché i suoi programmi non possono accedere a nient'altro che alla memoria virtuale del processo che li esegue.

Quando una entità è presente in memoria virtuale significa che è presente in memoria fisica una tabella delle pagine contenente un descrittore, con bit *P*=1, il cui campo indirizzo contiene l'indirizzo fisico dell'entità in questione?

No, questo vuol dire che l'entità (o, almeno, la prima pagina che contiene l'entità stessa) è *presente* in memoria *fisica*.

Supponiamo che si verifichi un page-fault all'indirizzo virtuale V, causato dalla non presenza, in memoria fisica, della tabella delle pagine $T(V)$. La tabella delle pagine verrà caricata in una pagina fisica (di indirizzo fisico) F. Poiché la tabella $T(V)$ deve essere anche residente in memoria virtuale (per poterla poi modificare via software), dovrà avere un indirizzo virtuale V'. Per rendere presente la tabella $T(V)$, dobbiamo aggiornare l'entrata opportuna della tabella $T(V')$, scrivendoci l'indirizzo F e ponendovi *P*=1. E' possibile che $T(V')$ non sia presente a sua volta?

Dipende da come è organizzato il sistema. In generale, è possibile, nel senso che sarebbe possibile (anche se molto complicato), costruire un nucleo che gestisca la paginazione di se stesso: stiamo infatti richiedendo che la routine di trasferimento gestisca la paginazione delle strutture dati da lei stessa utilizzate. In genere, però, non ne vale la pena. Anche in questo caso, basta pensare al fatto che serve

una sola tabella $T(V')$ per mappare in memoria virtuale tutte le 1024 tabelle delle pagine, quindi possiamo tenerla sempre in memoria fisica, senza tanti problemi. Nel nucleo che abbiamo realizzato (scaricabile dal sito) è stata fatta una scelta diversa, ma che porta comunque a non avere mai page fault su $T(V')$. In pratica, una volta che $T(V)$ è stata caricata in memoria fisica all'indirizzo fisico F, si sceglie $V' = F$. Poiché la memoria fisica è mappata permanentemente in memoria virtuale, $T(F)$ è sicuramente presente. Se decidiamo di prendere la strada difficile (nucleo auto-paginato), dobbiamo ovviamente caricare anche la tabella $T(V')$ (e così via!).

La spiegazione dell'algoritmo di rimpiazzamento del TLB dice che l'aggiornamento del bit *B0* del campo *R* di un insieme, avviene in base al gruppo rimpiazzato o acceduto con successo, all'interno dell'insieme stesso, settando o resettando tale bit, rispettivamente se l'operazione ha coinvolto *L0* o *L1*, oppure *L2* o *L3*. Non sono sicuro di aver capito come avviene l'aggiornamento dei bit *B2* e *B3*.

I tre bit mi devono dire quale entrata della linea dovrà essere rimpiazzata la prossima volta che ci sarà un fallimento. Per avvicinarci alla politica LRU, dobbiamo fare in modo che l'entrata appena caricata venga scelta di nuovo il più tardi possibile (essendo quella acceduta più di recente). Supponiamo, per esempio, di aver appena caricato *L1: porremo B0*=1 (in modo che, al prossimo fallimento, si vada a pescare nell'altra coppia) e *B1*=0 (in modo che, quando si dovrà scegliere nuovamente dalla prima coppia, si sceglierà *L0*). Non modifichiamo *B2* (il bit che ordina le entrate *L2* e *L3*), in quanto non sappiamo niente dell'ordine con cui *L2* e *L3* erano state caricate.

Non riesco a capire che relazione abbia la routine di trasferimento con il TLB? Tutte le operazioni sul TLB non avvengono via hardware?

Infatti è così (a parte l'invalidazione selettiva o totale del TLB, che può essere ordinata via software: quella selettiva, tramite l'istruzione `INVLPG`, mentre quella totale scrivendo in *CR3*). La routine di trasferimento, che è software, entra nel discorso perché è interessata al valore del bit D di ogni pagina (tale valore le serve a decidere se la pagina, qualora venisse scelta come vittima di un rimpiazzamento, debba essere preventivamente salvata in memoria di massa).

Riguardo al TLB il trasferimento del descrittore di pagina, viene effettuato da una routine o via hardware? (Se non ci sono istruzioni per accedere al TLB, come fa, la routine, a trasferirvi il descrittore?)

Ovviamente hardware. Tale trasferimento fa parte delle normali azioni eseguite dal microprogramma che gestisce i fallimenti nel TLB.

Poniamo vi sia una tabella delle pagine A, mappata in memoria virtuale da una pagina P puntata da una diversa tabella delle pagine B. Se non si modifica A, non c'è nessun bisogno di accedere a P, quindi il bit Accessed riferito ad P rimane a 0. La routine di rimpiazzamento a questo punto potrebbe scegliere di rimpiazzare P, ma così facendo toglierebbe dalla memoria fisica la tabella delle pagine A, senza che nel direttorio il bit Present ad essa riferita venga messo a 0 (e quindi, non verrebbe generato page-fault). Così non può funzionare. Dove sta l'errore?

L'errore sta nel non tenere conto che bisogna organizzare il sistema in modo che ciò che lei descrive non accada. Nel sistema illustrato sopra, la pagina virtuale P, che mappa in memoria virtuale la tabella A, sarà quella il cui indirizzo virtuale coincide con l'indirizzo fisico della tabella A. In altre parole, il sistema accede alla tabella A (e a tutte le tabelle) tramite la memoria fisica mappata in memoria virtuale (la famosa *finestra* di cui parlo nella FAQ). Questo implica che la tabella B sarà una delle tabelle dedicate a questo mapping e, come spiegato nella FAQ, il bit di presenza della pagina P, nella tabella B, varrà sempre 1.

Per ogni tabella delle pagine che si trova in memoria fisica, ci sono due descrittori: un descrittore di tabella, nel direttorio ed un descrittore di pagina, nella tabella che la fa risiedere in memoria virtuale. Quando viene generato un indirizzo virtuale, la traduzione coinvolge una tabella delle pagine ed il bit *A* del descrittore di tale tabella viene settato (via hardware) dalla MMU, nel descrittore all'interno del direttorio. Quando invece, si ha un accesso via software alla tabella (cioè si vuole accedere ad essa e non ad una delle pagine i cui descrittori sono contenuti al suo interno), il bit *A* che viene settato, è quello nel descrittore relativo alla pagina virtuale occupata dalla tabella stessa (che si trova nella tabella che mappa in memoria virtuale tutte le altre tabelle).

Esattamente.

Ad ogni pagina, residente in memoria fisica, è associata una variabile, usata per gestire il rimpiazzamento delle pagine secondo una data politica, che viene aggiornata periodicamente da una routine messa in esecuzione dal timer, che compie l'aggiornamento basandosi sul valore del bit *A* del descrittore della pagina. Per una tabella, quale bit *A* viene usato? Quello nel descrittore di tabella, nel direttorio o quello nel descrittore di pagina (nella tabella apposita)?

Quello nel direttorio. L'altro non serve a niente (solo che l'hardware non può saperlo).

Perché esiste la possibilità che alcuni descrittori possano avere permanentemente il bit *P* a 0 (implicando una minore dimensione della tabella di corrispondenza rispetto ai 4MiB+4KiB)?

Questo è proprio il motivo per cui abbiamo (anzi, l'Intel ha) deciso di paginare la tabella delle pagine: in questo modo, non è necessario che l'intera tabella

associata al processo corrente sia presente in memoria. Infatti, un processo qualsiasi, normalmente, non ha bisogno dell'intero spazio di 4 GiB di memoria. Con la tabella paginata, possiamo evitare di avere in memoria fisica le tabelle che mapperebbero le porzioni inutilizzate di tale spazio.

Perché nella situazione iniziale oltre al direttorio sono presenti in memoria fisica anche le tabelle delle pagine che consentono di posizionare il direttorio? Questo non è riferibile tramite *CR3*?

Perché vogliamo che il direttorio (e le tabelle associate) siano modificabili via software, perché la routine di rimpiazzamento è una routine software. Lei deve immaginare uno schema del genere:

Tutti gli indirizzi generati da CPU (prelevamento istruzioni, prelevamento eventuali operandi in memoria, scrittura di eventuali operandi in memoria) vengono tradotti da MMU (sono quindi virtuali). Il contenuto di *CR3* è l'indirizzo fisico del direttorio, chiamiamolo F. La CPU può sapere quanto è F, ma non ha alcun modo di raggiungere la locazione di memoria RAM F se nessun indirizzo virtuale viene tradotto in F. Ad esempio, supponiamo di scrivere un pezzo della routine di rimpiazzamento, usando *CR3* come suggerisce lei. Per prima cosa, la routine cercherà di leggere il descrittore associato all'indirizzo non tradotto:

```
   ...
   movl %cr2, %ecx
   andl $0xFFC00000, %ecx      # selezioniamo i 10 bit più significativi
   shrl $22, %ecx
1: movl %cr3, %ebx             # tentiamo di leggere il descrittore di tabella
2: movl (%ebx, %ecx, 4), %eax
   ...
```

L'indirizzo generato dall'instruzione 2 verrà tradotto da MMU, prima di raggiungere la RAM. È vero che sappiamo benissimo (istruzione 1) qual è l'indirizzo fisico del direttorio, ma, da software, non possiamo generare indirizzi fisici. Quell'indirizzo verrà tradotto come tutti gli altri. Dobbiamo fare in modo, allora, che esista un indirizzo virtuale V che, tramite la tabella, corrisponda all'indirizzo fisico F del direttorio, ed usare V per accedere al direttorio.

Che cosa si intende per indirizzo virtuale assoluto?

Un indirizzo (virtuale o non virtuale che sia) è assoluto quando riferisce direttamente una locazione di memoria, e relativo quando, per riferire una locazione di memoria, deve essere prima sommato ad un riferimento. Ad esempio, la seguente istruzione:

```
movl 100, %eax
```

contiene l'indirizzo (virtuale) assoluto 100. Nel seguente frammento di codice, invece:

```
ciclo:
   ...
   jmp ciclo
```

l'indirizzo che l'assemblatore e il collegatore scriveranno al posto dell'etichetta 'ciclo' nell'istruzione `JMP` è relativo, perché il formato dell'istruzione `JMP` prevede che il suo argomento sia lo spiazzamento (in byte) del target del salto, rispetto all'istruzione successiva alla `JMP`.

Cos'è la memoria lineare?

È il nome che, nell'architettura Intel, viene dato alla memoria virtuale realizzata tramite la paginazione, per distinguerlo dalla memoria logica, che è segmentata.

Pagina 20, volume IV: *il meccanismo di corrispondenza [...] richiede due accessi in memoria fisica.* Non sarebbero invece necessario 4 accessi, due per prelevare gli indirizzi, e due per porre ad 1 il bit *A* dei descrittori?

Sì, è vero. Comunque, i due accessi per porre i bit A (del descrittore di tabella e del descrittore di pagina) ad 1 vanno fatti solo se, nella precedente lettura, non risultavano già pari ad 1.

La routine di rimpiazzamento è la stessa sia per la segmentazione che per la paginazione?

Decisamente no. Fanno cose completamente diverse e sono associate ad eccezioni di tipo diverso (page-fault per la paginazione, segment-fault per la segmentazione).

È meglio realizzare una memoria virtuale tramite la segmentazione o tramite la paginazione?

Per quanto riguarda la memoria virtuale (cioè, creare l'illusione di una memoria fisica più grande di quella effettivamente a disposizione), l'arma vincente è decisamente la paginazione. La paginazione non ha il problema della frammentazione esterna (al prezzo di una piccola frammentazione interna). Inoltre, anche in un sistema segmentato conviene aggiungere la paginazione: in questo modo possiamo evitare di caricare/salvare i segmenti per intero e, quindi, possiamo anche avere segmenti più grandi della memoria fisica.

Dicendo che la corrispondenza tra pagine virtuali e pagine fisiche è a carico del sistema operativo s'intende che è il sistema operativo a riempire per ogni programma il contenuto dei descrittori di tabella e di pagina?

Sì.

C'entra qualcosa col fatto che il livello di privilegio delle tabelle delle pagine è sempre *S*?

Sì e no. Se si riferisce al bit S/U che si trova nel descrittore di tabella delle pagine, no, non c'entra niente. Per come il descrittore di tabella è stato presentato a lezione, solo i bit P e A hanno significato. Tutti gli altri bit del byte di accesso del descrittore di tabella non hanno alcuna utilità. Il nucleo "riempie" le tabelle via software e, quindi, come sappiamo, è necessario che le tabelle stesse siano in memoria virtuale. Poniamo che la tabella T si trovi all'indirizzo virtuale V. Quando il nucleo cerca di accedere via software alla tabella T, la MMU dovrà tradurre l'indirizzo V, passando da un descrittore di tabella nel direttorio e arrivando ad un certo descrittore di pagina in un'altra tabella T' (in genere diversa da T). È il bit S/U di *questa* entrata (nella tabella T') che deve essere S, affinché, via software, si possa accedere alla tabella T solo a livello di privilegio sistema.

Quando una tabella delle pagine viene scritta via software, viene aggiornato il bit *D* contenuto del descrittore di pagina che fa risiedere la tabella delle pagine in memoria virtuale. Di questa modifica ne tiene conto la routine di rimpiazzamento oppure questa rimpiazza la tabella in questione anche se il bit *D* contenuto del descrittore di pagina che fa risiedere quella tabella delle pagine in memoria virtuale è 1?

Dipende. Se ne parla nella FAQ. Comunque, tenete sempre presente che, per poter rimpiazzare una tabella delle pagine, questa deve essere "vuota": tutti i bit P devono essere uguali a 0. A questo punto chiediamoci: ci interessa salvare su memoria secondaria il contenuto di questa tabella? dipende da cosa abbiamo scritto nelle entrate con bit P=0 e se ciò che vi abbiamo scritto può essere diverso da quanto si trova già scritto nella copia della tabella che già si trova in memoria di massa. Si va da un estremo in cui nelle entrate con bit P=0 non scriviamo mai niente (in un tale sistema potremmo buttar via la tabella e, anzi, potremmo anche evitare di averne una copia in memoria di massa), passando per una situazione (che è quella sottintesa nel libro del prof. Frosini) in cui nelle entrate con P=0 scriviamo la posizione su memoria secondaria della corrispondente pagina, e questa posizione non cambia mai (in questo sistema possiamo evitare di ricopiare la tabella in memoria di massa, in quanto la copia che abbiamo in memoria principale è uguale a quella che si trova in memoria di massa) ad un altro estremo in cui, in ogni entrata, scriviamo ancora la posizione in memoria di massa della corrispondente pagina, ma questa posizione può cambiare ad ogni rimpiazzamento (in questo caso, dobbiamo salvare la nuova versione della tabella).

Non ho capito se la modifica software di una tabella delle pagine viene tenuta presente in fase di rimpiazzamento e inoltre che senso ha una modifica hardware della medesima tabella se poi non può essere notificata a nessuno (se il bit *D* del descrittore di tabella nel direttorio è privo di significato).

Le uniche modifiche alla tabella che la MMU esegue via hardware sono quelle ai bit A e D. Entrambi hanno senso solo per le entrate della tabella con il bit

P=1. Siccome, per poter rimpiazzare una tabella, questa deve avere tutti i bit P uguali a 0, non ha importanza sapere se i bit A e D sono stati modificati via hardware, tanto, a quel punto, non sono più significativi.

Tutte le tabelle delle pagine devono essere in memoria virtuale?

Non necessariamente, ma supponiamo di sì.

Se alcune di esse non sono in memoria fisica cosa contengono i descrittori di pagina che le fanno risiedere in memoria virtuale?

Se la posizione in memoria di massa della tabella l'abbiamo già scritta nella corrispondente entrata del direttorio, niente.

Pagina 13, volume IV: "*In presenza di memoria virtuale, la fase di rilocazione di un programma non si rende più necessaria: poiché per ogni programma viene predisposta un'opportuna tabella di corrispondenza, la memoria virtuale può risultare sempre disponibile a partire dall'indirizzo zero*". Non mi è molto chiaro questo passaggio.

Il punto è che quando scriviamo un programma in linguaggio macchina (o quando compilatore, assemblatore e linker lo scrivono per noi), dobbiamo decidere a quale indirizzo di memoria dovrà trovarsi ogni istruzione ed ogni dato del nostro programma. Quando, ad esempio, scriviamo qualcosa del genere:

```
int i;
int main()
{
        ...
        i = 1;
        ...
}
```

qualcuno deve decidere a che indirizzo deve trovarsi la variabile `i`, in modo che poi si possa tradurre l'assegnamento `i = 1` con

```
movl $1, <indirizzo di i>
```

Poniamo di aver scelto l'indirizzo 1000. Che succede se, nel momento in cui vogliamo caricare il nostro programma per eseguirlo, ci accorgiamo che l'indirizzo 1000 è già occupato da qualche altro programma (qualcun'altro potrebbe aver scritto un suo programma e aver scelto l'indirizzo 1000 per metterci una sua variabile, e magari il suo programma è in esecuzione proprio mentre cerchiamo di caricare anche il nostro)? Se abbiamo scritto 1000 dentro quella `MOVL`, il nostro programma funziona solo se `i` si trova all'indirizzo 1000, ma 1000 lo sta già usando quell'altro programma. Per risolvere il problema, facciamo così: scegliamo un indirizzo qualsiasi per `i` (qualcosa nella `MOVL` la dobbiamo pur scrivere), ma associamo al nostro programma eseguibile una tabella in cui annotiamo che, per eseguire il nostro programma, c'è da scegliere un indirizzo per `i` (e per tutto le altre variabili e istruzioni) che sia libero al momento del caricamento. Inoltre, la tabella dirà che, una volta scelto, l'indirizzo va scritto in quella `MOVL` (e in

tutte le altre istruzioni del nostro programma che usano `i`). Questa operazione (che è la rilocazione), verrà eseguita dal caricatore. Molto spesso, possiamo fare in modo che basti scegliere solo l'indirizzo di partenza del programma (l'indirizzo del primo byte in cui viene copiato l'eseguibile), invece che l'indirizzo di ogni variabile. Ad esempio, possiamo preparare il nostro eseguibile in modo che `i` si trovi ad un certo spiazzamento dall'inizio del file, ad esempio 100 byte, e tradurre l'assegnamento con:

```
movl $1, 100
```

Dopodiché, nella tabella di rilocazione, specifichiamo che il file va copiato in memoria a partire da un indirizzo libero, e questo indirizzo, una volta scelto, va *sommato* all'indirizzo che già si trova nella `MOVL` (questo è ciò che intende dire la frase contenuta nel libro Architettura dei calcolatori). Ovviamente, possiamo evitare tutta questa operazione di rilocazione se siamo sicuri che, ogni volta che vorremo eseguire il nostro programma, l'indirizzo 1000 (o comunque quello che abbiamo scelto) sarà sicuramente libero. La memoria virtuale ci offre questa sicurezza: ogni programma viene eseguito in un uno spazio di indirizzamento (virtuale) completamente a sua disposizione. Se anche qualcun'altro avesse scelto l'indirizzo 1000 per metterci una sua variabile, e i nostri due programmi venissero eseguiti comtemporaneamente, il sistema avrà cura di mappare il suo 1000 virtuale ad un indirizzo fisico diverso dal nostro 1000 virtuale.

Abbiamo detto che nella paginazione il trasferimento di una pagina da memoria di massa a memoria fisica può rendere necessario un trasferimento inverso. Quando però precisamente?

Se (e solo se) la memoria fisica è piena si rende necessario eliminare una pagina virtuale V_1 per far posto alla pagina entrante V_2. Non sempre, però, è necessario che la pagina eliminata venga anche ricopiata in memoria di massa: poiché V_1 era stata, precedentemente, anch'essa caricata in memoria fisica dalla memoria di massa, in memoria di massa è sempre presente una copia di V_1. Dobbiamo ricopiare V_1 in memoria di massa se (e solo se) V_1 è stata successivamente modificata, mentre si trovava in memoria fisica, perché in questo caso la copia che si trova in memoria di massa contiene dati "vecchi".

È vero che il direttorio delle pagina si trova sempre sia in memoria logica che fisica?

Sul mio onore.

La tabella delle pagine si trova sempre in memoria logica?

Quale tabella delle pagine? per ogni processo, ci sono potenzialmente 1024 tabelle delle pagine. Comunque, se una tabella non si trova in memoria logica, non può essere modificata dal processo (neanche quando il processo esegue a livello di privilegio sistema, a meno che la tabella non si trovi in qualche altra memoria logica del cui direttorio il sistema conosce l'indirizzo fisico). Poiché la realizzazione della memoria virtuale si basa sulla modifica, a tempo di esecuzione, delle tabelle delle pagine, almeno quelle in uso devono essere in memoria logica.

Bit *P* del byte di accesso del descrittore di pagina: è vero che viene messo ad 1 dalla routine di trasferimento ed a 0 da quella di rimpiazzamento?

Per le pagine gestite come memoria virtuale, sì. Le pagine che mappano la memoria fisica in memoria virtuale hanno sempre P=1.

Chi modifica i bit *R/W*, *S/U*, *PWT*, *PCD*?

La routine di trasferimento li inizializza ad un certo valore, in base a ciò che contiene la pagina (codice, dati, ...) e poi mantengono sempre quel valore finché P vale 1 (se la routine di rimpiazzamento pone P=0, quei bit non hanno più significato)

Bit *D:* è posto ad 1 dalla MMU via HW quando una pagina viene modificata. Posso quindi affermare che una volta ad 1 il suo valore rimane tale e non torna più a 0?

Servirebbe a ben poco se fosse così. Ogni volta che il contenuto della pagina viene ricopiato in memoria di massa, il software (in questo caso, la routine di rimpiazzamento) può rimettere D=0.

Se la memoria fisica è mappata in memoria virtuale a partire dall'indirizzo 0 fino all'indirizzo V solo la memoria virtuale da V in poi viene gestita normalmente (con *P* a 0 o a 1). Però nel testo si dice che ogni processo ha a disposizione tutti gli indirizzi a partire dall'indirizzo 0 (e ciò rende superflua la rilocazione). Dunque quel processo in realtà allocherà le sue variabili a indirizzi superiori all'indirizzo V?

Sì (se la finestra parte da 0).

Quindi la rilocazione deve essere effettuata?

No. Visto che si sa a priori quanto è V, il programma può essere direttamente collegato in modo che vada ad occupare lo spazio da V in poi. Non è necessaria una rilocazione al momento del caricamento. Tale rilocazione si renderebbe necessaria, invece, se tutti i programmi andassero caricati nello stesso spazio di indirizzamento, ad indirizzi non assegnati a-priori (come in MS-DOS): in quel caso, non possiamo sapere, al momento del collegamento del programma, a quale indirizzo il programma stesso verrà poi caricato per essere eseguito (e, in più, tale indirizzo cambierà da esecuzione ad esecuzione, a seconda di quali altri programmi si troveranno già in memoria in quel momento).

In caso di paginazione delle tabelle di corrispondenza noi sappiamo che si può avere page-fault sia se P=0 nel descrittore di tabella delle pagine o nel descrittore di pagina. Se ho P=0 nel descrittore di tabella delle pagine ho un page-fault e per mezzo dell'indirizzo fisico nel descrittore all'interno del direttorio carico la tabella delle pagine dalla memoria di massa. Sul libro si dice che tale tabella non ha niente di significativo al suo interno o più precisamente non riferisce alcuna pagina. Ha tutti i bit P=0. Gli indirizzi di pagina non sono significativi. L'istruzione che ha generato il page-fault nel direttorio viene rieseguita e questa volta mi da un page-fault nella tabella delle pagine. Da dove prelevo tale pagina in memoria di massa se l'indirizzo nel descrittore è non significativo?

Questa parte del sistema è lasciata un po' nel vago nel libro e in tutto il corso. Ovviamente, l'informazione a cui vi riferite (dove si trovano le pagine in memoria di massa) deve essere disponibile da qualche parte. Non è possibile dare una risposta precisa, restando nei limiti del sistema descritto nel corso: è necessaria una struttura dati aggiuntiva che mantenga questa informazione. Un sistema (teorico) semplice prevederebbe un campo in più, dedicato a mantenere questa informazione, per ogni entrata del direttorio e delle tabelle delle pagine. Purtroppo tale campo non è previsto nelle strutture dati dei processori Intel, a cui facciamo riferimento. Nel libro si accenna ad una soluzione (usare ogni entrata come una unione che contiene l'indirizzo in memoria di massa quando P=0, e byte di accesso + indirizzo fisico quando P=1), che però andrebbe specificata meglio e resa consistente con tutto il resto (si veda, ad esempio, la FAQ). Nei sistemi operativi reali per processori x86, è normalmente prevista una struttura dati a parte per questo scopo (parte di questa struttura è data dal file system, che nel corso non viene trattato).

Possiamo ipotizzare che ai descrittori di pagina che fanno risiedere la memoria fisica in memoria virtuale ci acceda solo la routine di rimpiazzamento per recuperare l'indirizzo fisico che deve utilizzare per l'aggiornamento del descrittore di pagina con P=0? Le chiedo questo perché nella FAQ lei ha parlato di una finestra per consentire un accesso speciale alla memoria fisica da parte del *codice di sistema*.

A dire il vero la routine di rimpiazzamento non ha alcun bisogno di accedere a quei descrittori per compiere l'operazione che lei dice: l'indirizzo fisico a cui lei si riferisce è ottenuto da una qualche struttura dati che tiene conto delle pagine fisiche ancora libere (se ve ne sono), oppure è l'indirizzo fisico che era associato alla pagina virtuale che è stata scelta per essere rimpiazzata, poniamo che sia la pagina V. È chiaro che V è fuori dalla *finestra* (è una pagina gestita tramite il meccanismo della memoria virtuale), quindi il suo descrittore non è tra quelli che *fanno risiedere la memoria fisica in memoria virtuale*. L'accesso speciale (in pratica un modo per by-passare la traduzione operata dalla MMU e accedere direttamente alla memoria fisica) è utile in molti casi, anche se non indispensabile:

- la struttura dati che tiene conto delle pagine fisiche libere (a cui accennavo prima) può essere realizzata tramite una lista, in cui ogni pagina fisica libera contiene, nei primi byte, un puntatore alla prossima pagina libera.

È chiaro che, per gestire tale lista, il sistema deve poter accedere a tutte le pagine fisiche, per poter leggere e scrivere tali puntatori;

- il sistema ha, in alcune occasioni, la necessità di allocare memoria, per se stesso, dinamicamente (ad esempio, nel sistema visto a lezione, per allocare le strutture `richiesta` nella primitiva `delay`(), ma anche i descrittori di processo e le pile possono essere allocate in questo modo). In questo caso, può essere comodo allocare tale memoria dallo stesso pool di pagine usate per la memoria virtuale: è evidente, però, che tale pool copre tutta la memoria fisica, quindi il sistema deve poter scrivere e leggere da tutte le pagine della memoria fisica, se vuole usare le strutture dati così allocate;

l'alternativa è *mappare al volo* le pagine fisiche a livello sistema, ogni volta che servono. Mappare a priori tutta la memoria fisica è, da questo punto di vista, una semplificazione.

A chi serve *CR3*? Alla MMU e basta? In quali occasioni?

Sì. Gli serve ogni volta che deve tradurre un indirizzo da virtuale a fisico. Considerato che l'unica cosa che fa la MMU è tradurre indirizzi da virtuale a fisico, direi che *CR3* gli serve sempre.

Come si fa a sapere se una pagina è rimpiazzabile o non può essere rimpiazzata?

La domanda è un po' vaga, una pagina può non essere rimpiazzabile per vari motivi:

- perché contiene un buffer che si sta usando per una operazione di I/O;
- perché contiene una tabella delle pagine non vuota;
- ...

La non rimpiazzabilità va vista caso per caso e, in generale, può essere stabilita a priori (ad esempio, si decide che un certo range di indirizzi è dedicato ai buffer di I/O) o necessita di strutture dati aggiuntive.

Se il bit *P* vale 1 non vuol dire semplicemente che la pagina è presente? Come fa a garantire cioè che la pagina non è rimpiazzabile?

Chi ha detto che il bit P=1 garantisce che la pagina non è rimpiazzabile? La routine di rimpiazzamento è software, quindi rimpiazza quello che il programmatore che l'ha scritta vuole che rimpiazzi. In questo caso, il programmatore non deve mai scegliere le pagine che mappano la memoria fisica come vittime di un rimpiazzamento (durante un rimpiazzamento, "vittima" è la pagina uscente, che viene eliminata per far posto alla pagina entrante). Per far ciò, visto che la memoria fisica è mappata agli indirizzi virtuali da 0 a $dimensione\_memoria\_fisica\_installata - 1$, gli basta cercare le pagine da rimpiazzare partendo da quest'ultimo indirizzo virtuale, a salire. Il bit P non c'entra.

Pagina 16, volume IV: *il direttorio [...] ha i 12 bit meno significativi uguali a 0.* 12 perché poco prima si e supposto che l'indirizzo di riga è a 12 bit?

Sì.

Fisicamente cio vuol dire che il direttorio comincia dal byte 0 della pagina? Se sì, è indispensabile.

Sì. Anche se non è indispensabile, ma l'hardware e il software sono molto più semplici in questo modo.

Potrebbe chiarirmi il paragrafo in cui si parla di rimpiazzamento di una tabella delle pagine?

Se una tabella viene scelta come vittima di un rimpiazzamento, sicuramente non riferirà nessuna pagina (questo perché il bit A associato alla tabella viene posto ad 1 ogni volta che viene posto ad 1 il bit A di una delle pagine da lei riferite, quindi una tabella non potrà, qualunque sia l'algoritmo di rimpiazzamento, avere mai un contatore minore di una delle sue pagine ma, al limite, uguale. Inoltre, abbiamo visto che, se i contatori sono uguali, viene sempre scelta una pagina. Questo ci assicura che, per poter essere scelta, una tabella deve essere per forza *vuota*). Se è vuota (tutti i bit P valgono 0) il suo contenuto e' non significativo, quindi possiamo evitare di ricopiarlo in memoria di massa.

D'altro canto, dobbiamo allora capire cosa succede nel caso inverso, quando, cioè, la tabella in questione dovesse essere ricaricata dalla memoria di massa alla memoria fisica. Anche in questo caso, la tabella, appena caricata, sarà certamente vuota, ed ogni entrata si presenterà come scritto nel paragrafo (bit P=0, indirizzo fisico non significativo).

Infine, l'ultimo paragrafo fa riferimento al fatto che la tabella di corrispondenza si trova essa stessa in memoria virtuale (in modo che possa essere riferita dalle routine di rimpiazzamento e del timer), come illustrato in figura 6.1.

Pagina 22, volume IV: "*Questo si verifica, per esempio, [...] il bit P passa da valore 1 a valore 0*". Sta parlando della pagina depositata in memoria di massa o di quella appena trasferita in memoria fisica? Se viene eseguito il rimpiazzamento il bit *P* non dovrebbe essere messo da qualcuno ad 1 proprio ad indicare che adesso la pagina c'è? Perché dovrebbe passare a 0?

Passa a 0 il bit P della pagina virtuale che viene ricopiata in memoria di massa (come scritto tra parentesi nel paragrafo in questione), ad indicare che quella pagina non risiede più in memoria fisica. Infatti, la pagina fisica, che la conteneva, ora contiene la pagina che è stata caricata, e il cui bit P è passato ad 1.

Pagina 23, volume IV: "*Notare che la possibilità [...] si mantiene nel passaggio da indirizzo virtuale a indirizzo fisico*". La frase si riferisce al fatto che ad ogni operazione il contenuto di *MAR* viene opportunamente incrementato? Ma se non si trasferiscono byte l'indirizzo viene incrememntato di 2 o 4; sono comunque da considerarsi indirizzi continui?

Sì.

Quale è la vera motivazione per cui a parità di condizione si sceglie di rimpiazzare una pagina e non una tabella delle pagine? Solo perché se si rimpiazzasse la tabella perderemmo ogni riferimento alle pagine da essa riferite (senza avere più la possibilità di accedervi via software) o c'è qualche altra motivazione più profonda?

No, è quella.

Potrebbe essere pure una motivazione di spazio? Per una pagina si rimpiazzerebbero 4KiB mentre per una tabella si rimp. 4MiB?

No, una tabella occupa 4KiB.

Se una tabella delle pagine al momento che viene caricata in memoria non riferisce alcuna pagina e il campo indirizzo è non significativo chi decide e gli assegna le pagine da riferire? Quando?

La routine di trasferimento. Quando si verifica un page-fault.

Come fa a sapere a quale tabella associare tali pagine?

Da *CR3* ricava l'indirizzo (fisico) del direttorio, dai dieci bit pi significativi dell'indirizzo che ha causato il page-fault (indirizzo salvato in *CR2*) ricava l'entrata del direttorio che contiene l'indirizzo (fisico) della tabella cercata. Infine, dai dieci bit *centrali* dell'indirizzo che ha causato page-fault ricava l'entrata, all'interno della tabella appena individuata, che deve essere modificata.

Cosa accadrebbe se nell'indirizzo per accedere al TLB scambiassi l'ordine dei campi etichetta e indice? Sorgerebbero dei problemi?

Continuerebbe a funzionare, ma ci sarebbero probabilmente più miss, in quanto molti descrittori di pagine consecutive andrebbero a finire nello stesso insieme, e quindi il TLB non ne potrebbe ricordare piuù di 2 contemporaneamente (per ogni insieme).

Noi abbiamo detto che portiamo la memoria fisica in memoria virtuale per poter accedere direttamente via software alla memoria fisica e questo quindi rappresenta un vantaggio. Se la memoria fisica fosse molto grande o addirittura uguale in dimensioni alla memoria lineare (4GiB) gli indirizzi virtuali disponibili sarebbero pochi o addirittura non ce ne sarebbero. Una soluzione sarebbe rendere più grande lo spazio di memoria lineare ma l'hardware a 32 bit della macchina non ce lo permetterebbe quindi sempre utilizzando un'architettura a 32 bit l'unica soluzione sarebbe di non mettere la memoria fisica in memoria virtuale. Quali sono le consegueze di una tale cosa? (sempre se il mio discorso abbia senso)

Il suo discorso ha perfettamente senso, anzi, quello a cui lei si riferisce è proprio il limite di questa tecnica. Ricordiamo, però, che questa tecnica è solo un modo per risolvere i seguenti problemi:

1. quando si attiva la paginazione, c'è un problema di continuità di indirizzamento: l'istruzione che segue quella che attiva la paginazione (quella che pone a 1 il bit 31 di *CR0*) deve avere lo stesso indirizzo prima e dopo dell'attivazione;
2. il codice e le strutture dati del nucleo (comprese quelle richieste dall'hardware, come *GDT*, *IDT*, direttori e tabelle, etc.) devono essere mappate in memoria virtuale, se vogliamo poter eseguire le primitive e permettere alle primitive di leggere/modificare le strutture dati stesse;
3. vogliamo che il nucleo possa allocare dinamicamente strutture dati, per suo uso interno (ad esempio, le strutture richiesta nel timer, o i descrittori di processo) e, possibilmente, che tali strutture possano essere allocate ovunque in memoria fisica (in modo da non imporre limiti artificiali alla memoria allocabile dal nucleo). Ovviamente, se poi il nucleo deve poter accedere ad una struttura dati così allocata, questa deve trovarsi in memoria virtuale.

Una tecnica che, tagliando la testa al toro, risolve, in modo semplice, tutti e tre i problemi, è appunto quella di mappare *tutta* la memoria fisica, a partire dall'indirizzo virtuale 0, nella memoria virtuale di *tutti* i processi (ovviamente, codice del nucleo, *GDT*, *IDT* etc. sono in memoria fisica, quindi, mappando tutta la memoria fisica, si mappano anche loro). È la soluzione adottata in Linux fino a qualche tempo fa (tranne per l'indirizzo virtuale di partenza, che non era 0, e il problema 1 veniva, e viene, risolto con un mapping temporaneo). Anche in Linux, quando i server hanno cominicato ad avere più di 3Gb di memoria, si è dovuta adottare un'altra tecnica. Quella realizzata nei kernel recenti funziona così:

- l'ultimo GiB di indirizzi virtuali di ogni processo è riservato al kernel;
- in quell'ultimo GiB, vengono permanentemente mappati i primi 896MiB di memoria fisica presente nel sistema (occupando, quindi, 896MiB dei 1024MiB di indirizzi disponibili per il kernel);
- se il kernel deve accedere alla memoria fisica oltre i primi 896MiB, crea un mapping temporaneo, usando gli indirizzi virtuali disponibili all'interno del suo GiB (quelli dopo i primi 896MiB);

- il kernel alloca la memoria (per i suoi usi interni) preferibilmente dai primi 896MiB di memoria fisica, mentre le pagine dei processi vengono caricate (preferibilmente), dopo i primi 896MiB.

In questo modo, ogni processo ha comunque 3GiB disponibili per la sua memoria virtuale, e, in molti casi, il kernel riesce ad allocare tutto ciò che gli serve in quei 896MiB.

Le eccezioni come sappiamo possono capitare in qualsiasi stato della fase di esecuzione di un'istruzione e memorizzano in pila come indirizzo di ritorno lo stesso indirizzo dell'istruzione che ha generato il fault. Nella fase iniziale di esecuzione è possibile che eventuali registri del processore siano stati modificati e quindi una successiva riesecuzione dell'istruzione porterebbe a risultati imprevisti. La soluzione che accenna il libro è quella di prevedere copie di lavoro dei registri del processore. Potrebbe spiegarmi meglio come vengono utilizzati tali registri aggiuntivi e quali problemi introducono a riguardo della velocità di esecuzione?

L'utilizzo è il seguente: ogni istruzione scrive esclusivamente nella copia di lavoro di ogni registro. Se una eccezione arriva durante la fase di esecuzione, il contenuto delle copie di lavoro viene semplicemente scartato (quindi è come se l'istruzione non fosse mai stata eseguita). Se, invece, l'esecuzione dell'istruzione arriva fino alla fine, il contenuto della copia di lavoro di ogni registro viene ricopiato nel registro corrispondente. L'impatto di questa operazione di ricopiatura sulle prestazioni è minimo o nullo (tutti i registri possono essere copiati in parallelo, e la copia può essere eseguita in parallelo con altre operazioni, come la decodifica dell'istruzione successiva).

Pagina 17, volume IV: "*la paginazione della tabella delle pagine [...] rende necessario che questa risieda a indirizzi virtuali diversi da quelli in cui risiede la memoria fisica*". Perché?

Chiamiamo *superpagina* una sequenza di indirizzi lineari che copre 4MiB, il cui primo indirizzo è multiplo di 4MiB (in pratica, immaginiamo i 4GiB di memoria lineare divisi in tronconi di 4MiB, a partire dall'indirizzo 0, e chiamiamo questi tronconi superpagine). Ogni superpagina è composta da 1024 pagine da 4KiB ciascuna. L'i-esima tabella delle pagine (quella puntata dall'i-esimo descrittore di tabella nel direttorio) si occupa di mappare l'i-esima superpagina in memoria fisica (più precisamente, le 1024 pagine dell'i-esima superpagina) In quel paragrafo, si vogliono mappare il direttorio e le tabelle delle pagine nel seguente modo:

- direttorio all'idirizzo $4\,GiB - 4\,MiB - 4\,KiB$ (in pratica nell'ultima pagina della penultima superpagina);
- tabella i-esima all'indirizzo $4\,GiB - 4\,MiB + 4\,KiB \cdot i$ (in pratica nella pagina i-esima dell'ultima superpagina)

In pratica, il direttorio e tutte le tabelle appaiono in memoria virtuale, una dopo l'altra, verso la fine degli indirizzi disponibili. In questo modo, quando

si vuole modificare un descrittore di tabella o di pagina, è facile sapere a che indirizzo virtuale andare a trovarlo. Bisogna anche avere cura, però, di rendere assenti (P=0 nel loro descrittore di pagina) le pagine virtuali che dovrebbero mappare quelle tabelle delle pagine che non sono presenti in memoria fisica. Siccome le pagine virtuali che mappano l'intera memoria fisica in memoria virtuale hanno tutte P=1, non possiamo usare gli stessi indirizzi virtuali per entrambe le cose.

Nella figura si mostra il caso in cui in memoria centrale siano presenti il direttorio e soltanto le tabelle necessarie a posizionarlo in memoria fisica. Si mostra un esempio in cui il direttorio e due tabelle delle pagine (chiamiamole 1 e 2, dall'alto in basso nello schema denominato "memoria fisica") sono presenti in memoria fisica. Le frecce a destra sono ottenute così:

- per mappare il direttorio nell'ultima pagina della penultima superpagina, serve una tabella delle pagine (la 1) puntata dalla penultima entrata del direttorio. L'ultima entrata di tale tabella deve puntare al direttorio;
- la tabella 1 è presente, quindi deve essere mappata nell'ultima superpagina. Serve quindi una tabella delle pagine (la 2) puntata dall'ultima entrata del direttorio. Poichè la tabella 1 mappa la penultima superpagina, deve essere mappata nella penultima pagina: la penultima entrata della tabella 2 deve puntare alla tabella 1 (qui, nella figura, c'è un errore);
- la tabella 2 è a sua volta presente. Poiché mappa l'ultima superpagina, deve essere mappata nell'ultima pagina. Quindi l'ultima entrata della tabella 2 punta alla tabella 2 stessa;
- la freccia interrotta non è un errore, ma una sorta di "e così via", per le altre tabelle delle pagine (supponendo che ce ne siano altre presenti).

Per quanto riguarda invece le frecce che vanno dalla memoria virtuale alla memoria fisica ogni freccia rappresenta l'indirizzo virtuale della pagina da cui parte, che viene spezzato in due: la parte superiore (i 10 bit più significativi) individua un'entrata del direttorio, mentre la parte inferiore (i 10 bit meno significativi) individuano un'entrata della corrispondente tabella delle pagine.

Nelle intenzioni del Prof. Frosini non c'era la volontà di mettere le tabelle delle pagine in ordine in memoria virtuale (nell'ultima superpagina). Il suo esempio voleva essere più generale, con le tabelle delle pagine in un ordine qualunque. Nella prossima versione del libro la figura sarà cambiata in modo che venga rispettato l'ordine naturale delle tabelle in memoria virtuale.

Questo tipo di mapping (che è soltanto un esempio) deve essere necessariamente privato poiché l'ultima pagina di indirizzi virtuali disponibili, per ogni processo, è riservata a tale tabella (fa da alias alla tabella). Non sarebbe possibile condividerla dato che nell'ultima pagina di indirizzi virtuali un processo ha la propria tabella (cioè l'ultima entrata del direttorio di ogni processo punta alla tabella che mappa tutte le tabelle delle pagine di quel processo) e quindi alle omologhe tabelle, relative agli altri processi, andrebbero assegnati differenti indirizzi virtuali e si cadrebbe nella situazione in cui, la stessa pagina fisica è mappata, da processi diversi, ad indirizzi virtuali diversi.

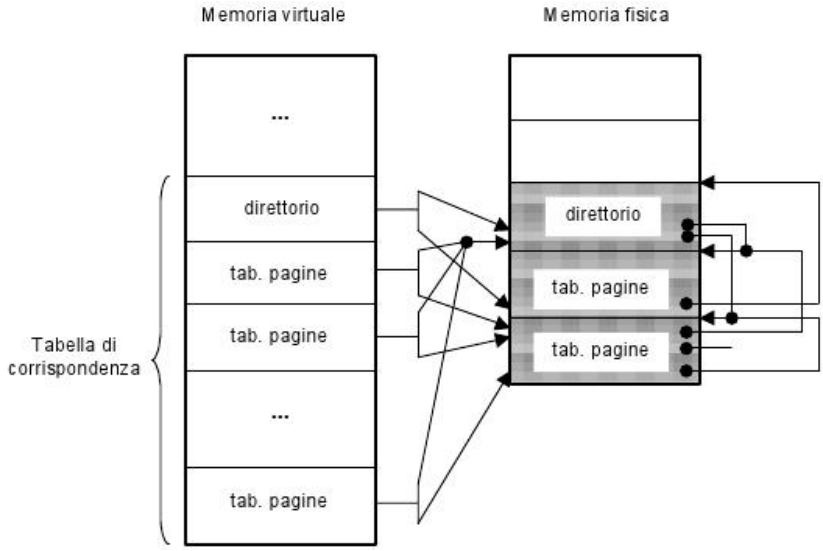


Figura 6.1: Tabella di corrispondenza in memoria virtuale

In base a quanto riportato nel libro e nella FAQ 5.1 si evince che la memoria virtuale gestita in maniera speciale (per realizzare la mappatura della memoria fisica in memoria virtuale) occupa la parte iniziale della memoria virtuale, quindi dall'indirizzo 0 ad un certo indirizzo *V*?

Non necessariamente. Se implementata, la mappatura può occupare un qualunque intervallo di indirizzi, purché identico in ogni processo. Ad esempio in vecchie versioni di Linux la mappatura partiva dall'indirizzo virtuale 0xC0000000 (corrispondente all'inizio dell'ultimo gigabyte). Se l'indirizzo di partenza è diverso da 0, però, bisogna ricordarsi che gli indirizzi lineari nella *finestra* non sono direttamente uguali agli indirizzi fisici, ma lo sono a meno di una costante (0xC0000000 in questo caso).

Quando è attiva la memoria virtuale i processi hanno lo stesso spazio di indirizzamento virtuale ma differenti spazi fisici. Se non si usa la memoria virtuale i processi farebbero uso dello stesso spazio logico che coinciderebbe con lo stesso spazio fisico?

Se non si usa neanche la segmentazione, sì.

Quindi la multiprogrammazione non potrebbe essere utilizzata perché non ci sarebbero tabelle di corrispondenza per *distribuire* le zone della memoria fisica ai vari processi?

Potrebbe essere utilizzata, eseguendo uno swap completo ad ogni commutazione di processo (come faceva CTTS, il primo sistema time-sharing) o caricando i processi in zone diverse della memoria fisica. Ovviamente, il primo modo è estremamente lento, mentre il secondo comporta una serie di problemi (i processi andrebbero collegati a partire da indirizzi diversi, oppure rilocati al momento del

caricamento, bisognerebbe affrontare il problema della protezione della memoria tra i processi, il problema della frammentazione etc.) per risolvere i quali sono nate le tecniche di segmentazione e paginazione.

In caso di paginazione attivata, le tabelle *IDT* e *GDT* sono anch'esse paginate, giusto?

Se la paginazione è attivata, in questo processore, tutto è *paginato*, nel senso che tutti gli indirizzi (tranne l'indirizzo del direttorio in *CR3* e gli indirizzi delle tabelle nel direttorio) vengono tradotti tramite il meccanismo della paginazione. Quindi, anche le tabelle *IDT* e *GDT* sono paginate.

Capitolo 7

Multiprogrammazione

Vorrei sapere dove risiedono le tabelle delle pagine e le pagine relative ad un processo nel momento in cui questo viene creato (quando in memoria fisica è presente solamente il direttorio e le tabelle delle pagine che permettono di posizionarlo in memoria virtuale).

Noi facciamo l'ipotesi che risiedano nell'area di swap (o memoria di massa).

Vorrei che mi illustrasse quali sono i problemi che portano all'invalidazione della memoria cache in caso di commutazione di contesto.

Non ce ne sono.

Pagina 28, volume IV: ***una pagina condivisa, quando ha al suo interno degli indirizzi assoluti, deve avere lo stesso indirizzo virtuale per tutti i processi.*** **Non riesco a capire bene.**

In quel paragrafo si suppone che due processi condividano una stessa pagina fisica, ma ad indirizzi virtuali diversi. Per esempio, pensiamo ad una pagina che contenga la traduzione del seguente frammento di assembler:

```
----------------------
inizio:
          ...
esempio:  .long ...
          ...
----------------------
```

Dove supponiamo che l'etichetta `inizio` corrisponda al primo byte della pagina in questione. Supponiamo, inoltre, che il dato `.long` ... si trovi a 500 byte dall'inizio della pagina (cioè, l'etichetta `esempio` sarà uguale ad `inizio` + 500). Sempre per esempio, supponiamo che la pagina si trovi in memoria fisica, a partire dall'indirizzo 4096, e sia condivisa da due processi, P_1 e P_2. Come spiegato nel libro, possiamo fare in modo che P_1 e P_2 usino indirizzi virtuali diversi per la stessa pagina. Per esempio, per P_1 l'etichetta `inizio` potrebbe valere 0 (e, di conseguenza, `esempio` varrebbe 500), mentre per P_2 `inizio` potrebbe valere 8192 (e quindi `esempio` sarebbe 8692). Tutto funziona, purché

nella tabella di corrispondenza di P_1 poniamo $0 \rightarrow 4096$ (l'indirizzo virtuale 0 si traduce nell'indirizzo fisico 4096) e nella tabella di P_2 poniamo $8192 \rightarrow 4096$. Vediamo ora cosa succede, però, se la pagina contiene un indirizzo assoluto che si riferisce (per semplicità) alla pagina stessa:

```
-----------------------------
inizio:
          ...
esempio:  .long ...
          ...
          movl esempio, %eax
          ...
-----------------------------
```

L'istruzione `movl esempio, %eax` contiene un indirizzo assoluto: l'etichetta `esempio` si riferisce direttamente all'indirizzo (virtuale) della variabile `.long ...` Una volta tradotto e collegato questo piccolo programma, l'etichetta `esempio` dovrà essere sostituita con un valore numerico preciso, ma quale? Il processo P_1 funziona se ci mettiamo 500, mentre il processo P_2 funziona se ci mettiamo 8692. Poiché stiamo assumendo che la pagina è condivisa, non possiamo metterceli entrambi, perché, in memoria fisica, avremo un'unica pagina che contiene la traduzione di quel programma.

Poniamo che il processo A sia in esecuzione e allochi in memoria fisica una pagina P che poi modifica (D=1). Avviene un cambiamento di contesto, ed entra in esecuzione il processo B (alla sua prima esecuzione, per esempio), che carica in *CR3* l'indirizzo fisico del suo direttorio. A parte il direttorio e le pagine necessarie per renderlo visibile in memoria virtuale, non risultano presenti in memoria fisica altre pagine dal punto di vista del processo B. Poniamo che ora B necessiti di caricare delle pagine in memoria fisica...

Prima di rispondere a queste domande, e alle successive, voglio fare una premessa: i meccanismi della traduzione degli indirizzi (e quindi la struttura dei descrittori di pagina, tabella etc.) sono realizzati nell'hardware del processore (normalmente, tramite un microprogramma). Su questi possiamo essere sicuri che funzionino nel modo descritto nel testo, e che non li possiamo cambiare (a meno di cambiare processore). Tutto il resto, invece (routine di trasferimento, rimpiazzamento, etc.) è *software*, e quindi, per definizione, modificabile. Nel testo, inoltre, viene detto solo a grandi linee cosa quelle routine devono fare, senza definire tutti i dettagli. Perché faccio questa premessa? perché la risposta più semplice che potrei dare alle sue domande è: "Mah! sono problemi del programmatore che deve scrivere quelle routine. Userà delle struttre dati fatte a posta". Quelli che elenca lei sono problemi di cui il programmatore deve tenere conto, e organizzare il sistema in modo che vengano risolti. Ci sono, ovviamente, molti modi diversi per farlo. Quindi, per rispondere nei dettagli alle sue domande, bisognerebbe fare riferimento ad un sistema ben preciso. Sistema che, però, non può che essere un esempio tra tanti, esistenti o possibili.

Detto questo, voglio comunque risponderle, facendo riferimento ad un esempio concreto di sistema (quello che abbiamo realizzato a scopo didattico, e che si può scaricare da `http://calcolatori.iet.unipi.it`):

- tutte le routine che gestiscono la memoria virtuale sono eseguite, da ogni processo, a livello sistema;
- ogni processo, quando si trova a livello sistema, può accedere a tutta la memoria fisica: la memoria fisica è mappata nella memoria virtuale di ogni processo, agli stessi indirizzi virtuali (cioè, a partire dall'indirizzo virtuale 0);
- ogni processo, quindi, quando si trova a livello sistema, può accedere a tutto ciò che si trova in memoria fisica. In particolare, può accedere alle tabelle delle pagine degli altri processi;
- inoltre, viene definita una struttura dati (accessibile da tutti i processi, ma solo da livello sistema) che contiene un descrittore per ogni pagina fisica del sistema. Il descrittore di una data pagina fisica ci dice se la pagina è libera o no. Se non è libera, ci dice (semplificando): da cosa è occupata (direttorio, tabella o pagina virtuale); se è rimpiazzabile o meno e, nel caso sia rimpiazzabile, quanto vale il contatore delle statistiche (quello aggiornato dalla routine del timer e utilizzato dalla routine di rimpiazzamento); da quale processo è occupata.

Questo schema è ispirato al modo in cui era organizzato Linux qualche tempo fa. La struttura dati analoga a quella di cui ho parlato nell'ultimo punto è ancora presente nei kernel recenti, mentre la gestione della memoria fisica in memoria virtuale è più complicata. Usando questo schema e questa struttura dati, si può vedere che i problemi da lei posti si risolvono facilmente.

... come fa B a sapere che deve salvare in memoria di massa la pagina P (modificata dal processo A e tutt'ora in memoria fisica) e non sovrascriverla semplicemente? ...

Basta leggere il bit D nella tabella delle pagine del processo A.

... Come fa B a sapere che quelli sono dati utili, visto che tale informazione sta in una tabella delle pagine del processo A (ovviamente, considerando il caso che tale tabella non sia condivisa dai due processi)?

C'è scritto nel descrittore della pagina fisica che contiene P.

I direttori dei vari processi devono essere sempre residenti in memoria fisica (P=1 sempre). È la routine di trasferimento che si occupa, trovata una pagina con P=1, di discernere se tale pagina sia rimpiazzabile o no?

Sì (più precisamente, la routine di rimpiazzamento), e lo fa leggendo il campo opportuno nei descrittori delle pagine fisiche.

Le parti della tabella di corrispondenza di un processo che devono sempre trovarsi in memoria fisica sono il direttorio delle pagine, la tabella che mappa il direttorio in memoria virtuale e la tabella che mappa tutte le tabelle delle pagine di quel processo (compresa se stessa e la tabella che mappa il direttorio), giusto?

Giusto (ma non solo quelle).

Page-fault: supponiamo di aver appena effettuato una commutazione di contesto. Avremo un *nuovo* direttorio (quello relativo al nuovo processo) e caricheremo (man mano che servono) in memoria tabelle delle pagine che all'inizio sono *vuote*. Ora, quando dobbiamo caricare una pagina da memoria di massa a memoria fisica, come fa questa routine a sapere quali zone di memoria fisica sono libere? Penso che effettui *direttamente* (cioè sempre tramite la *finestra* in memoria virtuale dato che è una routine software) un controllo sulla memoria fisica perché penso che non bastino le informazioni sul direttorio e sulle tabelle delle pagine. Ma, ripeto, come fa a capire che quell'area di memoria è rimpiazzabile o libera?

Ha bisogno di una struttura dati apposita. Ad esempio, potrebbe mantenere una lista di pagine vuote, in cui ogni pagina contiene, nei primi byte, l'indirizzo della prossima pagina libera. Se la lista è vuota, si deve rimpiazzare qualche pagina. All'inzio la lista comprende tutta la memoria fisica non usata per altri scopi (ad esempio per contenere il codice del nucleo e le sue strutture dati, come i descrittori di I/O e di processo), si svuoterà man mano che i programmi causeranno dei page fault, e si riempirà di nuovo ogni volta che un programma termina (liberando di conseguenza tutte le sue pagine).

Dopo una commutazione di contesto, cambiamo direttorio (modifichiamo *CR3*), ma le pagine appartenenti al vecchio processo che erano state caricate in memoria fisica rimangono inizialmente lì, giusto?

Sì.

Ora, come vengono considerate queste zone di memoria fisica in cui sono presenti pagine di un altro processo non attualmente in esecuzione? Libere o altro?

Ovviamente occupate.

Come facciamo a sapere se quelle pagine sono state modificate e che quindi, prima di essere sostituite con nuove pagine, vanno ricopiate in memoria di massa? Questa informazione non c'è data solo da un descrittore di pagina (bit *D* del byte di accesso) che sta in una tabella delle pagine che per adesso non è riferibile perché *appartenente* a un altro direttorio?

Perché non è riferibile? Non confonda ciò che è riferibile dalla MMU con ciò che è riferibile dal software: il direttorio puntato da *CR3* è quello usato dalla MMU per tradurre gli indirizzi, ma il fatto che un direttorio sia puntato da *CR3* non è

né necessario, né sufficiente a far sì che il software lo possa riferire. Affinché un direttorio (o una tabella delle pagine) possa essere riferito dal software è necessario che sia *mappato* in memoria virtuale, cioè che la traduzione correntemente attiva (quella puntata da *CR3*) possegga degli indirizzi virtuali che portano al direttorio. Cerco di fare chiarezza:

- ogni direttorio D_i è, di per se, una pagina fisica, che si trova ad un certo indirizzo fisico, poniamo F_i;
- ogni direttorio D_i (con le sue tabelle delle pagine), è anche una funzione da indirizzi virtuali a fisici, chiamiamola G_i. La funzione G_i è scritta nel direttorio D_i (e nelle tabelle), ma è attiva solo quando $CR3 = F_i$;
- la traduzione G_i avviene *sotto i piedi* del software, per opera della MMU: tutti gli accessi in memoria, ad un qualunque indirizzo V, vengono prima tradotti in $G_i(V)$. Quando è attiva G_i, il software può solo accedere alla parte di memoria fisica che si trova nel codominio di G_i;
- supponiamo che la traduzione correntemente attiva sia G_i e che il software vuole accedere al direttorio D_k. Per poterlo fare, la pagina fisica che contiene D_k deve appartenere al codominio di G_i, cioé deve esistere un indirizzo virtuale V_k tale che $G_i(V_k) = F_k$. Notate che, però, $CR3 = F_i$, che non c'entra assolutamente niente con F_k.

Quindi, anche se è il processo P_i a generare un page-fault, e la traduzione attiva è G_i, la routine di rimpiazzamento può accedere anche ad un direttorio diverso, D_k, purchè D_k sia raggiungibile tramite la traduzione G_i ad un qualche indirizzo V_k. La famosa finestra ci aiuta anche in questo caso: la finestra è tale che, per ogni i e per ogni $V < F_{max}$, $G_i(V) = V$. Quindi, se scegliamo $V_k = F_k$, raggiungiamo il direttorio D_k.

Tutti i direttori dei vari processi che potranno andare in esecuzione devono risiedere nella memoria fisica. Ma come è possibile sapere quanti sono i possibili processi che vanno in esecuzione? È giusto dire che il direttorio viene caricato quando viene mandato in esecuzione un processo *e poi* resta in memoria fisica per tutta la durata dell'esecuzione del processo associato, mentre le varie tabelle delle pagine e pagine vengono rimpiazzate o meno a seconda delle richieste del processo in esecuzione?

Sì

Quando il processo è terminato completamente, la zona dove risiedeva il suo direttorio viene usata dagli altri processi per allocare in memoria fisica le loro informazioni?

Certo.

Ma come è possibile che due processi diversi tramite il meccanismo di interruzione vadano a richiamare lo stesso gate? Questi avrebbero anche lo stesso ID, cosa che che non è possibile perché esso identifica un solo processo.

Infatti non sarebbero due processi diversi, ma lo stesso processo. Semplicemente, quello stesso processo risponderebbe a più di un tipo di interruzione.

I direttori dei vari processi stanno sempre in memoria fisica qualunque sia il processo sia in esecuzione. Si può dire allora che le pagine fisiche che li contengono sono condivise fra i vari processi e fanno parte dello spazio di indirizzamento *comune* dei vari processi?

Sono condivise, ma non per il fatto di trovarsi in memoria fisica. Sono condivise per il fatto di risiedere nello spazio di indirizzamento di tutti i processi.

Come si fa ad evitare che un processo rimpiazzi, in memoria fisica, una delle pagine contenenti un direttorio di un altro processo?

Come per altre domande di questo tipo, potrei semplicemente rispondere "lei come farebbe?". Il problema è software, e il programmatore deve trovare qualche soluzione. In questo caso, potrebbe riservare ai direttori una parte della memoria fisica su cui la routine di rimpiazzamento non va ad agire, oppure avere una struttura dati a parte che dice qual è il contenuto di ogni pagina fisica (che è la soluzione adottata nel nucleo scaricabile dal sito), in modo che la routine di rimpiazzamento, nella sua ricerca, salti le pagine che contengono i direttori.

Pagina 28 volume IV: "*un identificatore ID (MULTIPLO DI 8) rappresentato su...*" Non è sbagliato? perché deve essere multiplo di 8?

Perché i 3 bit meno significativi di ogni selettore di segmento hanno un significato a parte (*GDT/LDT* e *RPL*).

Pagina 31, volume IV: "*la commutazione è effettuata con una sequenza di azioni analoga a quella prevista per il meccanismo di interruzione, ma il bit NT viene azzerato prima che il contenuto del registro EFLAG del processore venga memorizzato nel descrittore del processo uscente*". La commutazione di cui si parla nel pezzo riportato, viene distinta da quella che avviene tramite interruzione. Che differenza c'è tra le due? Sono due meccanismi distinti?

La differenza è che la prima viene innescata da una interruzione (hardware o software che sia), mentre la seconda viene innescata tramite l'esecuzione di una istruzione `IRET` (se, al momento dell'esecuzione, il flag *NT* del registro dei flag vale 1). L'idea è che un processo interrompa un altro, ma che poi si *ritorni* al processo che era stato interrotto. Il meccanismo è simile sia nell'interruzione che nel ritorno, ma non identico. Per esempio (come scritto nel libro), nell'interruzione il microprogramma controlla che il processo entrante *non* sia busy,

mentre nel ritorno controlla che *sia* busy (appunto perché deve essere un processo precedentemente interrotto). Ovviamente, un'altra differenza si trova nella gestione opposta del flag *NT*.

Ho un domanda riguardante la coppia `INT $gate_task` e `IRET`. Tali istruzioni salvano in pila *EFLAG*, *CS*, *EIP* come le int di interrupt, anche se le informazioni salvate sono già nel *TSS*?

No. Quando il gate è di tipo task, l'istruzione `INT` esegue una commutazione hardware, come descritta nel libro, che non ha niente a che fare con il suo *normale* comportamento.

Nel momento in cui c'è una commutazione inversa, il campo link del processo entrante (quello che era bloccato) rimane inalterato?

Sì

Qual è il motivo per cui il bit *NT* del processo terminato viene salvato a zero nel relativo descrittore di processo?

Cosa intende per processo "terminato"? credo di aver capito che si riferisce al processo uscente, chiamiamolo P_1. P_1 non è affatto terminato: semplicemente (tramite l'esecuzione della istruzione `IRET`) ha volontariamente lasciato il processore al processo entrante (quello precedentemente bloccato, chiamiamolo P_2). P_1 potrà riandare in esecuzione in futuro.

Il flag NT serve a ricordare che il processo attualmente in esecuzione (ad es. P_1) ne ha interrotto un'altro (ad es. P_2). Vogliamo ricordarci che P_1 è annidato, in modo da rimettere a posto le cose (ovvero, rimettere in esecuzione P_2), appena P_1 esegue una `IRET` (tramite la quale P_1 ci comunica che *per il momento ha finito*). Dopo che la `IRET` ha risistemato le cose così come erano prima dell'interruzione, P_1 non deve restare *marchiato* come processo annidato: ormai non lo è più. Per questo azzeriamo NT prima di salvare $EFLAG$ nel descrittore di P_1. Se non lo azzerassimo, ogni volta che P_1 venisse messo in esecuzione, risulterebbe ancora annidato. (Nota: con il meccanismo di commutazione hardware presentato nel libro, non c'è altro modo di rimettere in esecuzione P_1 se non *annidandolo*, quindi tutto il discorso su NT sembra ozioso. In realtà, nel processore Intel, si possono eseguire anche commutazioni di contesto hardware che non comportano annidamento, ma queste sono state omesse dal libro, credo per semplicità).

Il descrittore del processo uscente A in figura 7.1 (che ha NT=0) è relativo a un processo terminato. Non dovrebbe scomparire dalla coda?

Il processo A non è affatto terminato. È stato interrotto da B, che a sua volta è stato interrotto da C. Il processo A ha il bit NT=0 perché, essendo il primo, non è annidato a nessun altro.

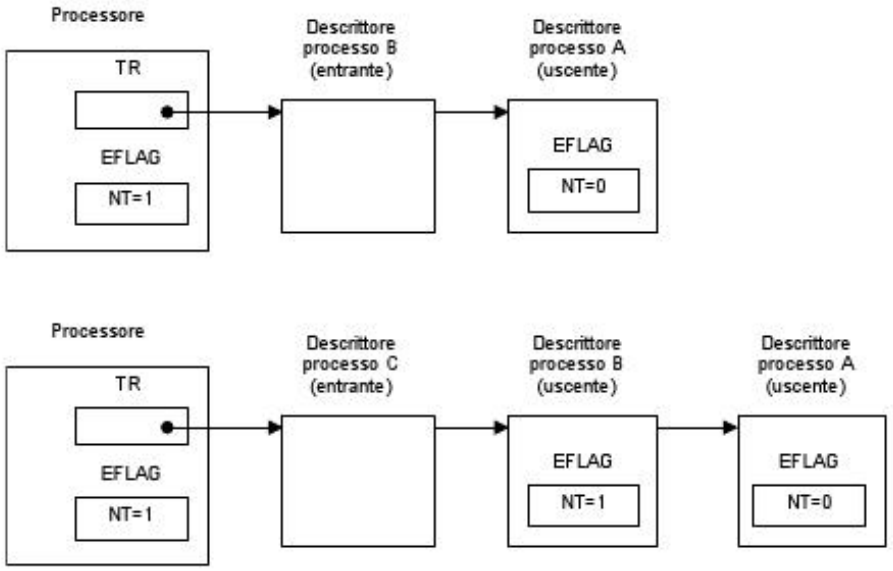


Figura 7.1: Annidamento dei processi

Al termine del meccanismo di commutazione hardware tra i processi viene esaminato il bit *NT*. Se *NT*=1 viene eseguita la commutazione (e fino qui tutto OK), altrimenti se *NT*=0 la commutazione non viene eseguita. Perché?

Credo si riferisca alla istruzione `IRET`. Se NT=0 vuol dire che la `IRET` non è relativa alla commutazione hardware. NT, infatti, viene posto ad 1 dal meccanismo di interruzione (quando il corrispondente gate è di tipo task), per ricordare che il task T_1, che sta andando in esecuzione, ne ha interotto un altro, T_2, e che T_2 deve tornare in esecuzione non appena T_1 esegue l'istruzione `IRET`. Però, mentre T_1 è in esecuzione, potrebbe essere interrotto nuovamente e, questa volta, il gate potrebbe essere di tipo interrupt/trap. Il processore passerà quindi ad eseguire la relativa routine di gestione dell'interruzione che terminerà anch'essa con una `IRET`. Questa `IRET` non deve rimettere in esecuzione T_2 tramite una commutazione hardware, ma T_1 tramite il semplice ritorno da interruzione (lettura di *CS*, *EIP*, *EFLAG* e, eventualmente, *SS* e *ESP* dalla pila corrente). Per distinguere i due casi, si usa il flag NT: il meccanismo di interrupt/trap salva *EFLAG* in pila e poi azzera NT.

In ambiente multitask, tutti i direttori dei processi e le tabelle che li individuano devono stare sempre in memoria fisica dalla creazione di ogni processo fino alla sua morte? Oppure quando cambio processo viene caricato in qualche modo il direttorio del processo entrante?

Per semplicità assumiamo che tutti i direttori siano sempre presenti in memoria fisica, per tutto il tempo di vita dei processi a cui appartengono.

Supponiamo che sia in esecuzione il processo P_1. Tale processo avrà caricati in memoria centrale codice e dati sensibili. Ad un certo istante giunge un cambio di contesto a favore del processo P_2: il contesto di P1 viene salvato, viene caricato il contesto relativo a P_2 e si carica il nuovo direttorio in *CR3*, cambiando di conseguenza spazio di indirizzamento. I bit di presenza saranno tutti settati a 0 e la routine di trasferimento provvederà *on demand* a caricare via via in memoria centrale le pagine che servono al processo P_2. Dal momento che si caricano pagine in memoria centrale non c'è il rischio di sovrascrivere dei dati che erano presenti al momento della commutazione di contesto tra P_1 e P_2 e quindi necessari a P_1? Se si, come può P_1 recuperare le informazioni andate perdute quando gli ritornerà il controllo? Avviene uno swap-back da ram a disco rigido ogni volta che si tenta di sovrasrivere una pagina in ram occupata? Come si può capire se tale sezione è occupata o libera?

La cosa è semplice. Il nucleo deve mantenere una struttura dati che gli permetta di stabilire se una data pagina fisica è libera o occupata. Questa struttura dati potrebbe avere la forma di una semplice lista che collega tutte le pagine libere (ogni pagina punta alla prossima libera), una bitmap (un bit per ogni pagina) o qualcosa di più complicato. Quindi, il problema che lei pone non c'è: le pagine di P_2 verranno allocate su pagine fisiche altrimenti libere. Se non ci sono pagine libere, ecco che interviene il rimpiazzamento: si sceglie in qualche modo una delle pagine occupate e la si libera, per far posto alla pagina richiesta da P_2. Ovviamente, può darsi che venga scelta una pagina che apparteneva a P_1. Se il sistema è in grado di sapere a chi appartiene una pagina fisica occupata (ciò può essere fatto prevedendo una ulteriore struttura dati che associa le pagine fisiche occupate al processo, e quindi al direttorio, che le mappa), deve anche eliminare dal direttorio/tabelle di P_1 il mapping per quella pagina (quindi, se P_1, tornato in esecuzione, ne avrà di nuovo bisogno, genererà un nuovo page fault e la pagina verrà ricaricata). Se il sistema non prevede strutture dati che gli permettano di associare una pagina fisica al suo proprietario, può procedere così: sceglie prima il processo a cui togliere una pagina, quindi scorre le pagine mappate dal direttorio/tabelle di quel processo alla ricerca della vittima. Quindi, le pagine di più processi possono tranquillamente convivere in memoria centrale. Infatti, non è necessario che P_2 parta con tutti i bit di presenza settati a 0, come dice lei. Ciò avviene solo quando P_2 parte per la prima volta, mentre, ogni volta che viene rischedulato, partirà con i bit a 1 come li aveva lasciati, meno quelli che sono stati rimessi a 0 perché qualche altro processo gli ha rubato le pagine mentre lui era bloccato.

Nella commutazione di contesto software la `carica_stato` provvede, fra le altre cose a mettere in *CR3* l'indirizzo fisico del direttorio associato al processo che sta per essere messo in esecuzione. *EIP* verrà (tramite la `IRET`) inizializzato con il valore che si trova nella pila sistema del processo entrante stesso; poiché in memoria fisica non sono ancora presenti le tabelle delle pagine verrà generato dalla MMU un page-fault. La routine di page fault provvederà a caricare la tabella delle pagine opportuna e aggiornerà un determinato descrittore di tabella nel direttorio (scrivendo e quindi modificando il direttorio). Se in memoria fisica non è ancora presente la tabella delle pagine che fa risiedere il direttorio in memoria virtuale (e nemmeno le tabelle delle pagine che mappano la memoria fisica in memoria virtuale), la MMU genera una nuova eccezione di page-fault che interrompe la routine di page fault stessa? Cosa succede di preciso?

Sì, la MMU genererà una nuova eccezione di page-fault interrompendo la routine di page-fault, nel momento in cui questa cerca di accedere al direttorio. Non ho capito se la sua curiosità deriva dal fatto che lei pensa che il direttorio (o la memoria fisica) vengano fatti risiedere in memoria virtuale dopo che la memoria virtuale è stata abilitata. In realtà, tutto viene predisposto prima (è la cosa più semplice). Quindi, il problema da lei posto non si verificherà mai (a meno di errori di programmazione nel nucleo).

Nel caso della commutazione hardware tra processi io pensavo che l'istruzione `IRET` che comporta una commutazione inversa tra processi facesse terminare il programma in esecuzione invece ho letto in una sua precedente risposta che ciò non è vero. Potrebbe spiegarmi meglio perchè? ...

La `IRET` rimette in esecuzione il processo collegato tramite il campo link del *TSS*, ma salva comunque lo stato del processo che esegue la `IRET`. Se quel processo venisse rimesso in esecuzione, ripartirebbe dal punto successivo alla `IRET`. Il meccanismo potrebbe essere usato per realizzare qualcosa di simile al meccanismo software handler/processo esterno. Anche qui, il processo esterno conterrebbe un ciclo infinito, con la `IRET` al posto della `wfi`() (non è così semplice, ovviamente, ma è giusto per dare l'idea).

...se non è così allora in che modo avviene la terminazione di un processo?

La *terminazione di un processo* è un evento che dipende fortemente dal sistema, e quindi dal software: in generale bisognerà deallocare le strutture dati associate al processo stesso (pile e descrittori, ad esempio), avvisare qualcuno che il processo è terminato (ad esempio, in Unix, viene avvisato il processo che aveva creato il processo ora terminato, in genere la shell), fare un po' di pulizie (ad esempio, chiudere i file che il processo aveva ancora aperti) ed altro. In genere, tali operazioni vengono eseguite da una primitiva apposita (`terminate_p`, nel nostro nucleo) che il processo stesso deve invocare (o che viene costretto ad invocare, se il sistema prevede qualche meccanismo per la terminazione forzata dei processi, come il task manager di Windows o il comando kill di Unix).

Supponiamo di lavorare in ambiente paginato e multitask. Supponiamo poi di avere dei processi attivati. Io ho in testa una visione della memoria centrale: una parte sarà riservata al sistema operativo, una parte allo scambio di parametri tra processi, il restante spazio sarà a disposizione per tutti i processi. Ognuno dei processi attivi (quando ha il controllo della cpu) vede l'ultimo spazio di cui parlo sopra come interamente disponibile per le proprie necessità. Sempre nel solito spazio saranno presenti tutti i direttori e tutte le pagine che li mappano in memoria. È giusto?

Più o meno, nel senso che le organizzazioni possibili sono tante e questa è una (ad esempio, questo spazio dedicato esplicitamente allo scambio di parametri tra processi non sempre è presente). Anche i direttori e le tabelle delle pagine, in genere, si trovano tutte nello spazio dedicato al sistema operativo.

Capitolo 8

Protezione

Quando si ha un'interruzione e si tiene conto anche della protezione vengono confrontati il bit *S/U* e *CPL*. Se *S/U* < *CLP* c'è un'eccezione di protezione. Mi potrebbe spiegare il motivo? Ovviamente non si potrebbe eseguire un programma utente perché non si sa cosa fa e il processore non può fidarsi delle sue azioni, ma passando a livello utente non sarebbero controllate l'esecuzione delle istruzioni privilegiate o l'accesso alle aree riservate producendo comunque eccezioni?

Il fatto è che, se ci troviamo a un livello di privilegio elevato (poniamo sistema), eseguiamo una `INT` e ci portiamo a un livello di privilegio più basso (poniamo utente) non possiamo essere sicuri che verrà eseguita la corrispondente `IRET`, in quanto, per definizione, non possiamo fidarci di ciò che avviene a un livello di privilegio inferiore.

Nel libro si dice che è prevista l'utilizzazione di due pile differenti, una per ogni stato di funzionamento del processore. Il fatto viene giustificato dicendo che, una sola pila, posta al livello utente, non sarebbe praticamente utilizzabile, per motivi di protezione, per informazioni riguardanti lo stato sistema, in quanto il prelievo dalla pila (operazione `POP`), non è distruttivo. Dunque, operazioni che operano sulla pila, in quanto tale, possono essere eseguite dal processore, solo se questo opera con livello di privilegio uguale a quello della pagina usata per realizzare la pila stessa. Quali sono i motivi di protezione che impediscono l'uso di una sola pila posta al livello utente? Forse, il fatto che le informazioni riguardanti lo stato sistema (anche una volta prelevate dalla pila), potrebbero essere accedute anche dal livello utente?

Questo è il motivo scritto sul libro (il *prelevamento* modifica solo *ESP*, non modifica in alcun modo il contenuto della pila). Il motivo riportato sui manuali Intel, invece, si riferisce al fatto che la *dimensione* della pila è critica per il funzionamento del sistema (deve esserci spazio per le informazioni che la CPU stessa vi deve scrivere) e non si può fare affidamento sullo spazio in cima alla pila utente. Si preferisce quindi passare ad un'altra pila, la cui dimensione è decisa dal programmatore di sistema e non è sotto il controllo del normale programmatore utente.

Spesso nel libro c'è scritto che le interruzioni hardware possono accedere ad un qualsiasi gate senza che vengano fatti controlli sul meccanismo di protezione. Quale è la vera motivazione? ...

Perché non si saprebbe che controllo fare. Quando una interruzione viene richiesta via software, è perché è stata scritta dentro un programma, questo programma è stato eseguito da un processo, e questo processo, al momento in cui ha eseguito l'istruzione `INT`, aveva un livello di privilegio, per cui ha senso chiedersi se quel livello di privilegio è ammissibile. Quando arriva una interruzione hardware, che livello di privilegio le assegniamo? e come? in questo processore, è come se il livello fosse *sistema*, d'ufficio, quindi è inutile controllare che vada bene (andrà sicuramente bene).

... Facendo così non c'è rischio di violare le regole di protezione?

Mah, dipende da quali sono le regole di protezione. In genere, il livello di privilegio dei gate di interruzione viene assegnato in modo che le normali primitive siano liberamente invocabili da tutti i livelli, mentre i driver/handler lo siano solo dal livello sistema. In questo modo si impedisce che, da livello utente, si possa invocare direttamente un driver/handler con una `INT` (cosa che causerebbe ogni genere di confusione nel sistema).

Quando si parla del cambiamento di stato, in seguito ad una interruzione di tipo interrupt/trap, si fa riferimento al salvataggio nella pila sistema dei valori di *EFLAG CPL* ed *EIP*, tuttavia nel momento in cui il processore compie questa azione ha ancora il valore di *CPL* a 0 e si trova quindi in stato utente. Come fa il processore a scrivere nella pila sistema che si trova in pagine a livello sistema se è in stato utente? Non si verifica una eccezione di protezione?

Un carceriere può entrare e uscire liberamente dal carcere? direi di sì, non vi pare? È il microprogramma del processore che impone il rispetto delle regole di accesso ai segmenti. È ovvio che il microprogramma stesso non è soggetto alle regole.

Pagina 36, volume IV: "*in conseguenza di quanto detto, pagine private di ciascun processo si possono avere, oltre a livello utente, anche a livello sistema*". Non capisco.

Un processo può avere pagine private sia a livello utente che a livello sistema (a livello sistema, saranno private le pagine che contengono la pila sistema di quel processo).

Nel caso di commutazione hardware o software che utilizzi gate interrupt è possibile avere una commutazione di contesto che ci porta da un processo utente ad un processo sistema oppure da uno sistema ad uno utente?

La commutazione di contesto è indipendente dai livelli di privilegio dei processi entranti e uscenti, quindi possono succedere entrambe le cose.

Capitolo 9

Sistemi multiprogrammati

Nella struttura di una primitiva, nel sottoprogramma d'interfaccia viene invocata la primitiva vera e propria (contenuta nel modulo `sistema.s`), la quale può provocare una commutazione di contesto e pertanto deve provvedere al salvataggio dello stato attuale con l'invocazione della `salva_stato`: tale sottoprogramma provvede a salvare anche la pila del processo nel suo descrittore, ma deve salvare il giusto *ESP*, poiché in testa alla pila ci sarà l'indirizzo di ritorno del sottoprogramma `salva_stato`: vorrei sapere se tale indirizzo contenuto in *EIP* si riferisce all'istruzione successiva alla chiamata della `salva_stato` nel modulo `sistema.s`?

All'inizio dell'esecuzione della `salva_stato`, in cima alla pila ci sarà chiaramente l'indirizzo dell'istruzione successiva alla `CALL salva_stato`, proprio per effetto dell'esecuzione dell'istruzione `CALL`. Subito sotto, ci sarà l'indirizzo successivo all'istruzione `INT $tipo` che ha portato, messo in pila dall'esecuzione dell'istruzione `INT` stessa.

Nel modulo `sistema.s`, la `carica_stato`, oltre ad altre operazioni, deve trasferire dalla vecchia pila sistema alla nuova pila sistema l'indirizzo di ritorno del sottoprogramma stesso, per garantire una corretta terminazione: qual è questo indirizzo di ritorno? L'istruzione successiva alla `CALL carica_stato` nel modulo `sistema.s`?

Certo.

La routine `inserimento_coda_timer`() dopo aver inserito l'elemento che vuole restare bloccato per un certo periodo nella coda suddetta, aggiorna il campo `d_attesa` dell'elemento successivo. Se non lo aggiorna cosa succederebbe? Perché aggiorna solo il successivo e non gli altri eventuali elementi successivi?

Perché in quella struttura dati, il descrittore $(i+1)$-esimo ricorda quanto tempo deve passare dopo il risveglio del descrittore i-esimo. Se l'elemento $(i+1)$-esimo contiene un tempo ΔT, vuol dire che, se il processo bloccato sull'elemento i-esimo si risveglia al tempo T_i, il processo bloccato sul descrittore $(i+1)$-esimo si risveglierà al tempo $T_{i+1} = T_i + \Delta T_{i+1}$. Se dobbiamo inserire un nuovo

elemento k tra i e $i+1$ (cioè se il processo associato a k dovrà svegliarsi ad un tempo T_k tale che $T_i < T_k < T_i + \Delta T_{i+1}$) e non aggiorniamo ΔT_{i+1}, il processo $(i+1)$-esimo si sveglierà a $T_k + \Delta T_{i+1}$, e non più a $T_i + \Delta T_{i+1}$. Però, una volta che abbiamo aggiustato ΔT_{i+1} in modo che T_{i+1} sia quello di prima, non c'è bisogno di aggiustare anche ΔT_{i+2} etc., perché questi fanno riferimento a T_{i+1} che non è cambiato.

Le strutture facenti parte del nucleo come le code dei processi devono stare nello spazio comune a livello sistema: possono essere soggette a rimpiazzamento? Perché?

(Quasi) tutto si può fare. Bisogna vedere quanto è complicato farlo e se ne vale la pena. In genere, siccome tali strutture dati, in un modo o nell'altro, sono usate anche dalla routine di page-fault, risulta estremamente complicato renderle rimpiazzabili (un page-fault sulle pagine che contengono tali strutture dati manderebbe in esecuzione la routine di page-fault che, però, avrebbe bisogno di quelle stesse strutture dati per porter funzionare). Nel nucleo illustrato nel libro (che è estremamente semplice) c'è una ulteriore limitazione: si assume che tutte le funzioni che operano su quelle strutture dati siano atomiche e tale atomicità viene ottenuta con la disabilitazione delle interruzioni mascherabili. Siccome, però, la routine di page-fault esegue operazioni di I/O, deve riabilitare le interruzioni. Non è quindi consentito che si verifichino page-fault mentre si accede alle strutture dati del nucleo (altrimenti, la riabilitazione delle interruzioni distruggerebbe l'atomicità).

Vorrei dei chiarimenti su come un processo sistema possa richiamare una primitiva di nucleo. Dalla figura 9.1 mi sembra di capire che anche un processo sistema per invocare una primitiva di nucleo deve utilizzare un sottoprogramma di interfaccia (il quale genera una `INT` che mette in esecuzione la primitiva di nucleo) ma non ne capisco il motivo. Non potrebbe semplicemente chiamarla senza fare uso del sottoprogramma di interfaccia dato che la primitiva di nucleo viene definita o nel file `sistema.cpp` (dove credo che stia anche il processo) o in `sistema.s` che viene comunque collegato dal linker con `sistema.cpp`?

Sì, può chiamarla con una semplice `CALL`, ma non deve trattarsi di una primitiva che può causare il blocco volontario del processo (come, per esempio, le primitive `sem_wait`, `sem_signal` o `delay`). Questo perché i processi bloccati verranno sempre rimessi in esecuzione tramite la coppia `CALL carica_stato`; `IRET`, quindi la pila sistema deve contenere le informazioni richieste dall'istruzione `IRET` (che coincidono con quelle inserite in pila dall'istruzione `INT`, ma sono diverse da quelle inserite in pila dall'istruzione `CALL`).

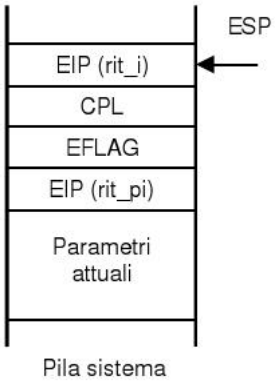


Figura 9.1: Chiamata di una primitiva di nucleo da parte di un processo sistema

Nella commutazione hardware che avviene attraverso i gate di tipo task è necessario che il descrittore di un processo memorizzi il valore di tutti i registri generali e di controllo (tra cui *EIP* ed *EFLAG*), il valore di *CR3*, quello di *CPL* e, successivamente anche *CPL* per discriminare il livello di privilegio del processo, infatti il meccanismo di interruzione si comporta diversamente nel caso di un gate di tipo task e non salva niente in pila, si limita a salvare e caricare il contenuto dai descrittori e poi termina con un microsalto alla fase di chiamata. La commutazione software invece è un meccanismo più sofisticato effettuato tramite primitive e driver. In tal caso la routine che causa la commutazione provvede a chiamare la `salva_stato` scrivendo nel descrittore del processo uscente e poi, dopo aver posizionato in esecuzione il nuovo `proc_elem` invoca la `carica_stato` ed esegue la `IRET`, tuttavia quando la routine comincia si trova in pila sistema i valori di *EFLAG*, *CPL* ed *EIP* (eventualmente anche il valore di *ESP* della pila utente) e salvando nel descrittore ESP (che punta in questo caso alla pila sistema) salva automaticamente anche il valore di *EIP*, *EFLAG* e *CPL* dato che questi sono in pila, infatti, come spiegato a pagina 56 del volume IV: "*i valori dei registri EFLAG, CPL ed EIP si trovano nella pila sistema del processo che ha invocato* `a_primitiva` *e non prevedono spazio apposito nel descrittore di processo*". Osservando però la struttura di `des_proc` a pagina 51 del volume IV, qua compare anche cpl, come mai? ...

Non c'è un vero motivo. Il descrittore di processo *dovrebbe* contenere tutte le informazioni associate al processo, quindi anche il suo livello di privilegio; è molto probabile che, in pratica, si possa fare a meno di quel campo.

... come si dovrebbe rispondere alla domanda: "che struttura ha un descrittore di processo?"

Va bene quanto si vede a pagina 51 del volume IV (per la commutazione software). I campi importanti sono gli altri, comunque.

Quando avviene una commutazione di contesto tramite gate di tipo interrupt, visto che avviene a livello sistema, in quale pila si salvano *EIP*, *CPL* e *EFLAG*?

Nella pila sistema (è un caso particolare della normale gestione di un interrupt)

Pagina 50, volume IV: "*l'istruzione IRET agisce sulla pila sistema del nuovo processo*". Perché?

Perché la `carica_stato` avrà caricato lo stato del processo entrante, che potrebbe essere diverso dal processo corrente. Lo stato del processo entrante comprende anche il valore dei registri *SS* e *ESP*, cioè il puntatore alla pila. La `IRET` agisce su qualunque cosa sia puntata da *SS:ESP*, quindi agirà sulla pila del nuovo processo.

Come si comporta di preciso lo schedulatore? Fa questo?

```
void schedulatore() {
        proc_elem* lav;
        rimozione_coda(pronti, lav);
        esecuzione = lav;
}
```

Sì, in pratica fa questo. Volevo però farvi notare che da http://calcolatori.iet.unipi.it (sezione approfondimenti) potete scaricare una versione completa e funzionante del nucleo, in cui tutti i dubbi di questo tipo possono essere risolti guardando il codice.

Lo schedulatore si cura o meno di ciò che è puntato da esecuzione? Dobbiamo sempre stare attenti noi a non perdere quello a cui punta esecuzione? Quindi dobbiamo chiamarlo solo dopo aver messo esecuzione in una qualche coda, come viene fatto nel nucleo del libro: tipicamente lo chiamiamo dopo la inspronti, o dopo l'inserimento nella coda del timer.

Esattamente. Lo schedulatore deve solo scegliere il prossimo processo da mettere in esecuzione. Ciò che accade al processo in esecuzione prima della chiamata allo schedulatore dipende dal punto in cui lo schedulatore stesso viene chiamato.

La inspronti invece fa questo?

```
void inspronti() {
        esecuzione->puntatore = pronti;
        pronti = esecuzione;
}
```

Sì, fa questo. Anche qui, vale quanto detto prima.

La `salva_stato`, per accedere al descrittore del processo in esecuzione, nel quale deve salvare i registri del processore, utilizza la parte nascosta di TR (base e indice) oppure l'*ID* che trova nel descrittore puntato da esecuzione e accede alla *GDT*? Dovrebbe usare *TR* in quanto il procedimento è più veloce? ...

La `salva_stato` non può usare la parte nascosta del registro *TR*, appunto perché è *nascosta*. In altre parole, il set di istruzioni del processore non prevede istruzioni per leggere o scrivere il contenuto di quei registri che si trovano nella parte nascosta. Quindi, non può che usare le informazioni contenute nella *GDT*. Potete guardare la funzione `des_p` nel file `sistema/sistema.S`, per vedere precisamente cosa si deve fare per ottenere un puntatore al descrittore di processo, dato il suo identificatore. Devo però mettervi in guardia su una cosa: i descrittori di segmento del processore vero sono un po' più complicati di quelli visti a lezione. Nella funzione `des_p` tale differenza viene nascosta nella macro `estrai_base`. Se non vi interessano tutti i dettagli del vero processore, basta sapere che tale macro si aspetta l'indirizzo di un descrittore di segmento in *EAX* e lascia la base del segmento stesso in *EBX*.

... infine, la `carica_stato` deve aggiornare tutto il registro *TR*? Perché anche se la `salva_stato` non lo utilizzasse, lo utilizza il meccanismo di interruzione quando avviene un cambio di privilegio, per accedere al puntatore pila sistema nel descrittore del processo in esecuzione.

Qui vale il discorso precedente. Da software non è possibile accedere alla parte nascosta dei registri selettore. La parte nascosta viene caricata automaticamente, come parte dell'esecuzione delle istruzioni che scrivono nei registri selettori: `LTR` per il registro *TR*, `MOVW qualcosa`, `%DX` etc. per i selettori di segmenti dati, `JMP` e `CALL` (nelle versioni "far"), `INT`, `RET` (versione "far") e `IRET` per il registro *CS*. Anche qui, vi rimando al codice della `salva_stato` e della `carica_stato`, in `sistema/sistema.S`.

Il meccanismo di commutazione hardware è pensato per salvare lo stato di un processo per farlo poi ripartire esattamente dal punto di salvataggio (indirizzo successivo alla `INT` che ha richiesto la commutazione, ad esempio). Invece, la coppia `salva_stato`, `carica_stato` è tale per cui il processo viene fatto ripartire scavalcando tutta la primitiva in cui il processo si era bloccato. Non ho ben capito questa affermazione, se un processo *ordina* una commutazione hardware con il comando `INT $gate_task_xx`, poi riparte dall'istruzione successiva alla `INT`, dov'è il problema?

Nessun problema in generale. Semplicemente, è diverso dal modo in cui è organizzato il nucleo visto a lezione. Per avere un comportamento il più simile possibile alla coppia `salva_stato`, `carica_stato`, bisognerebbe aver cura di chiamare la `INT` sempre alla fine delle primitive, mai nel mezzo.

In ogni caso, il problema principale per cui sarebbe complicato usare la commutazione hardware (almeno la forma con la coppia `INT`/`IRET`, che è l'unica che avete visto) è che induce un comportamento a "pila" (per via della lista di processi interrotti che si viene a formare tramite i campi *LINK*) che non si sposa con la schedulazione più generale che può avvenire nel nucleo. Per

esempio, può benissimo accadere che un processo P_1 si blocchi su un semaforo di sincronizzazione, quindi vada in esecuzione P_2, che si blocca a sua volta su un altro semaforo di sincronizzazione, e manda in esecuzione P_3. Infine, P_1 viene risvegliato (magari da un driver) e fa preemption su P_3. Con il meccanismo `INT/IRET` hardware, questa sequenza non può realizzarsi, perché non è a pila (P_2 sarebbe dovuto tornare in esecuzione prima di P_1).

Il prelievo dei parametri da passare ad alcune primitive (tali parametri vanno ricopiati in cima alla pila di sistema), può avvenire anche dalla pila utente (se il processo che ha invocato la primitiva ha, come livello di privilegio proprio, quello utente), senza violazione delle regole di privilegio, in quanto, al livello sistema, la pila utente, è contenuta in pagine dati. Perché non si ha violazione? L'idea che ho io è la seguente (ma non sono sicuro): la violazione non c'è perché l'operazione di ricopiamento dei parametri non agisce sulla pila come tale (tipo push o pop), ma è semplicemente un'operazione di lettura di dati, per eseguire la quale, il processore deve trovarsi in stato di privilegio maggiore od uguale a quello della pagina cui sta accedendo (e non soltanto uguale).

Esattamente (la copia avviene tramite `MOV`).

Le funzioni utilizzate per la scrittura dei driver di I/O (ad esempio `halt_input()`, `inserimento_coda()`, `rimozione_coda()`, `schedulatore()`) che tipo di primitive sono?

Non sono *primitive di nucleo*, ma semplici funzioni. Per i nostri scopi, le primitive di nucleo sono esclusivamente quelle funzioni invocate tramite il meccanismo di chiamata delle primitive (che prevede un interrupt software). Le primitive di nucleo sono invocabili da livello utente e comportano un innalzamento del livello di privilegio del processore. Le funzioni che lei cita sono semplici funzioni ad uso interno del nucleo. Quindi:

- non possono essere invocate da livello utente;
- all'interno del nucleo, sono invocate tramite normali istruzioni `CALL`.

Non mi torna la posizione del bit *BUSY* nell'istruzione 0 e non mi riesce proprio capire cosa facciano le istruzioni dalla 1 alla 7 nel codice seguente:

```
carica_stato:

    xorl   %ebx, %ebx
    movl   _esecuzione, %edx
    movw   (%edx), %bx # esecuzione->identifier in ebx
    pushl  %ebx

    shrl   $3, %ebx
    movl   gdt_ptr, %edx
```

```
    leal   (%edx, %ebx, 8), %eax # ind. entrata della gdt relativa in eax
0:  andl   $0xfffffdff, 4(%eax)  # bit busy del tss a zero

    popl   %ebx
    ltr    %bx

    movl   (%eax), %ebx
    movl   4(%eax), %edx
1:  shrl   $16, %ebx
2:  movl   %edx, %eax
3:  andl   $0xff, %eax
4:  shll   $16, %eax
5:  orl    %eax, %ebx
6:  andl   $0xff000000, %edx
7:  orl    %edx, %ebx            # base del tss in %ebx

    ...
```

Il descrittore di segmento visto a lezione è semplificato rispetto a quello vero dei processori Intel. Quello vero è così:

- primi 4 byte:
  - bit da 0 a 15: bit 0–15 del limite;
  - bit da 16 a 31: bit 0–15 della base;
- secondi 4 byte:
  - bit da 0 a 7: bit 16–23 della base;
  - bit da 8 a 16: byte di accesso;
  - —
  - bit da 16 a 19: bit 16–19 del limite;
  - bit 20 a 22: non ci interessano;
  - bit 23: G (granularità);
  - bit da 24 a 31: bit 24–31 della base.

Perché hanno fatto questo pastrocchio? Semplice: la segmentazione l'hanno introdotta nel 286, che era un processore con registri a 16 bit e 24 piedini di indirizzi. Nel 286, il descrittore di segmento finiva dove ho messo la linea (era su 6 byte, seguito da due byte di allineamento non significativi). Se guardate i primi 6 byte, la base risulta di 24 bit (quanti erano i piedini per gli indirizzi) e il limite di 16 bit (cioè la dimensione di un registro). La granularità era solo di byte, quindi un segmento poteva essere grande al massimo 64k e, per accedere a tutto lo spazio fisico disponibile (16Mb), si dovevano usare per forza più segmenti. Nel 386 sono passati a 32 bit ed hanno esteso il descrittore di segmento, ma, per restare compatibili con il 286, hanno dovuto arrangiare i pezzi in quel modo (per fortuna avevano previsto quei due byte di allineamento). Quella parte di codice della `salva_stato` prende i vari pezzi e ricostruisce la base e il limite.

Nella struttura `proc_elem` è presente un campo `short identifier`. Questo campo è lo stesso che poi è in `des_proc`, `shortid`? ...

Che vuol dire è lo stesso? `proc_elem` è una struttura che definisce un elemento in una coda di processi. Il campo `identifier` serve a specificare *quale* processo si trova in quella posizione della coda (posizione rappresentata da una variabile di tipo `proc_elem`). Da una parte avremo l'insieme delle variabili di tipo `des_proc`, ognuna con un valore diverso nel campo `id`, dall'altra un insieme di code, costruite con variabili di tipo `proc_elem`. Ogni variabile di tipo `proc_elem` conterrà, nel campo `identifier`, l'identificatore di un qualche processo, vale a dire un valore uguale al campo `id` di una qualche variabile `des_proc`.

... Oppure in `des_proc` questo campo è quello che poi verrà definito come il campo *LINK*?

No, il campo *LINK* si trova nei segmenti di tipo *TSS*. Il segmento di tipo *TSS* è una struttura dati definita dall'hardware del processore.

Il descrittore di processo dovrebbe contenere tra le altre cose un campo precedenza, ma questo non compare da nessuna parte. Come mai?

È un errore. Il campo precedenza è stato spostato da `des_proc` a `proc_elem` (dove ora si chiama `priority`). Concettualmente non cambia niente: si tratta, in entrambi i casi, di associare una priorità ad ogni processo.

Quando un processo passa da stato utente a stato sistema, il suo codice rimane quello memorizzato nella sezione testo dello spazio comune utente?

Questa non l'ho capita. A stato sistema ci può passare solo tramite il meccanismo di interruzione, che fa saltare il flusso di controllo all'entry point della routine di gestione di quella interruzione. Se, nel far questo, il livello di privilegio diventa *sistema*, vuol dire che tale routine si trova in un segmento codice di livello *sistema* (e, quindi, nello spazio comune a livello sistema).

Lo spazio comune per i processi utente è in comunicazione con lo spazio comune per i processi sistema?

Che vuol dire "in comunicazione"? Comunque si tratta di due spazi distinti.

Due processi a livello di privilegio diverso possono condividere qualcosa oppure, anche se hanno lo stesso codice, questo è memorizzato in spazi comuni diversi?

Possono condividere sicuramente dei dati (nella segmentazione, basta metterli in un segmento che si trova al livello di privilegio minore tra i due, mentre nella paginazione basta metterli in pagine a livello di privilegio utente). Ad esempio, i buffer usati per i trasferimenti dai dispositivi sono condivisi in questo modo. Nella segmentazione, è anche possibile condividere codice: è necessario, però, (a meno di non usare segmenti alias) usare un segmento codice di tipo *conforme*. Nella paginazione, l'unico modo di condividere codice tra livello

utente e livello sistema è quello di usare pagine alias (ad esempio, supponiamo di avere una routine che entra in una sola pagina: la mappiamo ad un certo indirizzo virtuale V_1, con bit S/U=1, e ad un altro indirizzo virtuale V_2, con bit S/U=0). In questo caso, però, il codice deve essere scritto in modo che possa essere usato contemporaneamente ad indirizzi virtuali diversi (cioè, non deve contenere indirizzi assoluti che puntino al suo interno).

Quando il processo attualmente in esecuzione viene reinserito nella coda dei processi pronti perché, durante l'esecuzione della `c_primitiva`, possa essere rischedulato, viene utilizzata una chiamata `CALL inspronti` nella `a_primitiva` prima della chiamata alla `c_primitiva`. Non si potrebbe evitare questa chiamata mettendo, nel corpo della `c_primitiva`, un `inserimento_coda(pronti, esecuzione)` dall'effetto equivalente?

Sì, ma la funzione `inspronti` è ottimizzata per quel caso particolare: poiché sappiamo che il processo in esecuzione è sicuramente quello a priorità maggiore rispetto a tutti quelli in coda pronti (altrimenti, non sarebbe in esecuzione), la `inspronti` inserisce direttamente in testa alla coda pronti.

Sul libro viene trattata per prima la commutazione di contesto via hardware e si dice che avviene mediante gate di tipo task. Più avanti si parla di commutazione software e si ipotizza che avvenga mediante una routine di sitema e con gate di tipo interrupt. In tal caso come prelevo l'identificatore del processo entrante visto che nel gate di tipo interrupt il campo ID non è significativo?

La commutazione software è gestita, appunto, via software. Il sistema definisce e gestisce delle strutture dati da cui estrae queste informazioni. In questo caso, si tratta della coda dei processi pronti: ogni volta che è necessario eseguire una commutazione di contesto, viene estratto un descrittore di processo da questa coda.

Nella commutazione a livello software che ruolo continuano a giocare i meccanismi hardware ed i gate di tipo task?

Nessuno. È un meccanismo che l'Intel rese disponibile 20 anni fa, credendo di fare cosa gradita, ma che nessuno usa (era usato nelle primissime versioni di Linux, ma ora non più). È troppo rigido e, essendo presente nei processori moderni solo per compatibilità con il 286, non è neanche ottimizzato (quindi, può risultare più lento della commutazione software).

Nella `sem_signal` viene gestita la prelazione *a mano*, non sarebbe meglio una soluzione del tipo seguente:

```
...
if(s->counter<=0) {
    proc_elem* app;
    rimozione_coda(s->pointer,app);
    inserisci_coda(pronti,esecuzione);
    inserisci_coda(pronti,app);
    schedulatore();
```

```
}
```

Dipende, meglio in base a quale parametro? Dal punto di vista didattico (che è l'unico che è stato considerato nella scrittura di quel nucleo) credo che la soluzione da lei proposta sia meno evidente. In particolare, è meno immediato far capire che in esecuzione ci andrà il processo che esegue la `sem_signal`, oppure quello risvegliato (e non qualche altro processo che si trova in coda pronti). Ovviamente, si tratta di opinioni: dal punto di vista funzionale, la sua soluzione è equivalente a quella del libro.

Come mai `desinti` e `desinto` vengono definiti in assembler?

Non c'è un motivo particolare.

Chi si occupa di mettere in qualche coda il processo uscente nel caso di commutazione di contesto causata da una `a_primitiva`? In questa pagina, in cui viene descritta una *a_primitiva* che provoca una commutazione di contesto, non viene detto chi mette in coda il processo uscente. Lo fa il sottoprogramma `salva_stato`,la `c_primitiva` prima di chiamare lo schedulatore, o lo fa direttamente la `a_primitiva` prima di chiamare la `c_primitiva`?

La sua domanda trova risposta andando a guardare le specifiche primitive che rientrano in questa classificazione, ovvero quelle che causano commutazione di contesto. In ogni caso, l'operazione viene svolta dalla `c_primitiva`. Ad esempio, la primitiva `sem_wait` (che è una tra quelle che possono causare una commutazione di contesto) esegue `inserimento_coda`(`s->pointer`, `esecuzione`).

Nel primo capitolo riguardo alla multiprogrammazione si fa riferimento ad un campo link presente nei descrittori di processo grazie al quale possiamo creare una catena di processi. Dal momento che si utilizzano strutture dati più complesse come le code dei processi, che utilità ha il campo link?

Se facciamo tutto via software (come nel nucleo mostrato nel libro), il campo link non ha alcuna utilità. Il campo link è usato dalla commutazione di contesto hardware. Poiché, però, in questo processore, la struttura del descrittore di processo è definita dall'hardware (che la usa anche in altre occasioni, come nel microprogramma di risposta alle interruzioni), dobbiamo prevedere lo spazio per il campo link, anche se non lo usiamo.

Ho letto da qualche parte che il meccanismo di commutazione hardware fra processi in realtà poi non e usato e che nella maggior parte dei sistemi operativi si implementa il tutto in via softwre. È vero? Che vantaggi ci sono?

Il meccanismo di commutazione hardware, almeno così come l'ha pensato l'Intel nel microprocessore 286 (perché di questo stiamo parlando) è troppo rigido e, visto che pochi lo usano, poco ottimizzato da parte dell'Intel stessa, che lo mantiene solo per compatibilità con il passato. Linux lo usava nelle primissime versioni, ma non lo usa più da molto tempo. Nel nucleo del prof.

Frosini, usare il meccanismo di commutazione hardware sarebbe molto problematico, in quanto tale meccanismo è pensato per salvare lo stato di un processo per farlo poi ripartire esattamente dal punto di salvataggio (indirizzo successivo alla INT che ha richiesto la commutazione, ad esempio). Invece, la coppia `salva_stato`/`carica_stato` è tale per cui il processo viene fatto ripartire scavalcando tutta la primitiva in cui il processo si era bloccato.

La gestione software può essere veloce tanto quanto la gestione hardware?

In un processore moderno (con cache interna, pipelining, parallelismo, branch prediction) sì, certo, soprattutto se la gestione hardware non è molto ottimizzata. Ad esempio, il manuale Intel dice chiaramente che l'istruzione `PUSHA` (che salva in pila tutti i registri) potrebbe essere più lenta della sequenza, equivalente, di `PUSHL %EAX`; `PUSHL %EBX` etc.

Pagina 37, volume IV: i"*un descrittore di processo contiene [...] il puntatore della pila utente costituito dal valore ESPv*". Nel quarto capito ho letto che se il processo è sistema ha solo pila sistema. Dunque nei descrittori di processi sistema il registro *ESPv* non esiste o non punta a nulla?

Esiste (perché la struttura del descrittore di processo è fissa, stabilita dall'hardware), ma contiene un valore non significativo.

Pagina 51, volume IV: "*id, destinato a contenere un identificatore di processo (tale identificatore si può riferire al processo che ha attivato il processo stesso, a quello che ha mandato il processo stesso in esecuzione etc.)*". Allora in pratica questo campo *id* è *il sostituto* del campo *link* della commutazione hardware? Se sì, allora il suo contenuto è diverso sempre e comunque dal contenuto del campo identifier dell'elemento `proc_elem` relativo al processo stesso?

Sì. In realtà, `des_proc` è una semplificazione didattica del segmento TSS e il campo `id` rappresenta il campo `link` che si trova all'inizio del TSS. Se guardate il nucleo effettivamente realizzato dal vostro collega (scaricabile dal sito http://calcolatori.iet.unipi.it) vedrete che, quando si va a realizzare davvero `des_proc`, si deve seguire fedelmente la struttura del TSS, anche se alcuni campi non vengono usati (come il campo id/link nel nostro caso), in quanto la struttura viene letta anche dall'hardware, anche se non usiamo la commutazione hardware (quando viene letta? pensate alle interruzioni...)

Alla domanda *cosa vuol dire creare un processo* cosa risponderebbe? Della creazione fa parte la creazione del descrittore e delle pile? Fa parte l'inizializzazione del descrittore stesso? Vi rientra l'eventuale creazione del corpo?

Sì, sì, no. Il *corpo*, ovvero il codice che il processo deve eseguire, viene creato durante la compilazione, non durante la creazione di un processo che esegue quel corpo. Un processo (che è un concetto astratto) potremmo identificarlo con le strutture dati che lo descrivono e che venono create espressamente per

quel processo (quindi, descrittore e pile). Siccome il corpo può essere comune a più processi, mi sembra più utile considerarlo un'entità a se stante.

Pagina 56, volume IV: "*[...] e quindi il trasferimento in ESP di un nuovo valore che si riferisce a un livello di riempimento della nuova pila così come determinato dal meccanismo di interruzione*". Cosa significa?

Senza la parte che la precede, niente. La frase (partendo dall'inizio) dice che non è necessario che la `a_primitiva` ripulisca la pila prima di chiamare la `carica_stato`, perché tanto la `carica_stato` sovrascriverà il valore di *ESP* (assieme a tutti gli altri registri).

Potrebbe commentarmi il codice relativo alla routine *inserimento_coda_timer*? Ho compreso il meccanismo di inserimento ma mi riesce comunque difficile seguire il codice pur disegnandomi le famose scatoline della memoria (alla Lo Priore).

`r` e `precedente` servono a scorrere la lista con la classica tecnica dei due puntatori. Il `while` termina quando siamo arrivati alla fine della lista (`r` diventa uguale a 0, cioè `NULL`) oppure abbiamo già fatto l'inserimento (tracciato dalla variabile booleana `ins`). Poiché dobbiamo fare in modo che il numero di intervalli di attesa di un certo elemento sia dato dalla somma dei campi `d_attesa` di tutti gli elementi che lo precedono e di quello dell'elemento stesso, nell'inserimento sottraiamo il valore di tutti i campi $r->$ `d_attesa` (che incontriamo scorrendo la lista dalla cima) dal campo $p->$ `d_attesa` dell'elemento che vogliamo inserire. Continuando ad andare avanti nella lista e a sottrarre, si arriva o alla fine della lista, o ad un punto in cui una ulteriore sottrazione ci porterebbe ad un campo $p->$ `d_attesa` negativo: a quel punto ci dobbiamo fermare (`ins` $=$ `true`, che ci farà uscire dal `while`) e lì dobbiamo inserire il nuovo elemento. All'uscita dal `while`, c'è una normale operazione di inserimento in una lista. (A proposito, il cognome è Lopriore).

L'istruzione `IRET` di una primitiva prevede che vengano estratti dalla pila sistema i valori dei registri *EIP*, *CPL*, *EFLAG; quando* un processo è nuovo chi ha provveduto a mettere nella sua pila opportuni valori di tali registri (che verranno poi prelevati dalla istruzione `IRET` della primitiva che per la prima volta lo metterà in esecuzione)? forse la `activate_p`?

Chi se no?

Che vuol dire che, usando le nostre primitive per la memoria dinamica, il numero di byte allocati viene associato alla struttura allocata? Cioè l'utente deve prevedere un campo nella propria struttura che contiene il numero di byte?

No.

Altrimenti dove viene salvata?

È un dettaglio implementativo delle funzioni `mem_alloc`/`mem_free` o degli operatori `new`/`delete`. Ad esempio, una semplice soluzione prevede che `mem_alloc(x)` allochi, in realtà, $x+4$ byte all'indirizzo, poniamo, $p-4$, e che restituisca al chiamante l'indirizzo p. Nei 4 byte *nascosti*, la `mem_alloc` scrive il numero x (cioè il numero di byte richiesti dal chiamante). Quando verrà invocata `mem_free(p)`, la funzione accederà all'indirizzo $p-4$ per sapere quanti sono i byte da deallocare.

In una interfaccia di conteggio a ciclo continuo la costante viene nuovamente caricata nel contatore da chi? dalla interfaccia stessa? Vvia software dal programmatore?

Dall'interfaccia stessa.

Nel corpo del driver del timer di sistema riportato di seguito, `p` non è inutile?

```
p=descrittore_timer;
descrittore_timer=descrittore_timer->p_rich;
delete p;
```

No, `p` serve a ricordare l'indirizzo che era contenuto in `descrittore_timer`, prima di modificarlo con l'assegnamento `descrittore_timer` = `descrittore_timer`$->$`p_rich`, in modo da poter poi eliminare la struttura `richiesta` che si trovava in cima alla lista (`delete p`).

Che significa che a seguito della esecuzione di una routine di interruzione la sua pila si trova esattamente nelle condizioni iniziali per cui la coppia SS:ESP non necessita di essere salvata nel suo descrittore?

Che quando la routine di interruzione sarà terminate, avrà riportato il livello della pila esattamente al livello a cui era all'inizio (se ha fatto delle `PUSH` o delle `CALL`, avrà fatto anche le corrispondenti `POP` e `RET`, ad esempio), quindi non ci sarà bisogno di salvare il valore attuale di *SS:ESP* nel descrittore: il valore attuale sarà uguale a quello già salvato nel descrittore.

Pagina 50, volume IV: i"*notare che tutti i processi hanno la sommità della pila sistema nella stessa situazione*". Che importanza ha?

È di fondamentale importanza, in quanto è proprio questo fatto che ci permette di passare da un processo qualsiasi ad un altro qualsiasi, usando sempre lo stesso codice per eseguire il passaggio (`carica_stato`/`salva_stato`).

Se durante l'esecuzione della `a_primitiva` non si è avuta commutazione di contesto alla fine si ritornerà al punto successivo rispetto alla invocazione della `a_primitiva` (dopo `INT $tipo`). Ma se si è avuta commutazione di contesto? Dove si ritorna dopo l'esecuzione della istruzione IRET della `a_primitiva`?

Al punto in cui il nuovo processo si era bloccato.

Perché il bit *B*, presente in ogni entrata della *GDT per* indicare se quel processo lo si può mandare in esecuzione o meno, non può stare in un gate dell'*IDT* invece che nelle entrate della *GDT*?

Perché ci potrebbe essere un altro task gate che punta allo stesso descrittore *TSS*. Quindi, se il bit *B* fosse nei task gate, ne potremmo avere più di uno per lo stesso *TSS*, con evidenti problemi di consistenza.

Nel libro si dice che i driver non possono richiamare le primitive del nucleo perché lo stato del processo in esecuzione, inizialmente salvato dal driver, verrebbe sovrascritto dalla nuova `salva_stato` della primitiva di nucleo. Non sarebbe più preciso dire che il driver non può chiamare solo le primitive di nucleo che possono causare una commutazione di contesto e non tutte le primitive di nucleo?

Sì, è così.

Perché un driver non può accedere alle primitive di nucleo?

Non può accedere a quelle primitive di nucleo che prevedono il salvataggio e il caricamento dello stato, in quanto, non essendo un processo, non ha un descrittore associato su cui salvare il proprio stato.

Capitolo 10

I/O

Pagina 84, volume IV: "*il buffer di memoria specificato da* P_1 *deve risiedere nello spazio di indirizzamento comune e non deve essere soggetto a rimpiazzamento dovendo essere utilizzato dal driver quando è in esecuzione* P_2". Quello che non capisco è come mai deve essere in una porzione non soggetta a rimpiazzamento, quando il professore ne ha parlato ha detto che questo derivava dal fatto che i driver (essendo routine di sistema) girano con le interruzioni disabilitate, non capisco cosa c'entri col meccanismo di rimpiazzamento. L'unica spiegazione plausibile che ho trovato è che, ponendo che il buffer possa essere rimpiazzato, potrebbe essere tirato fuori e dentro dall'area di swap e in questo modo, anche se i suoi indirizzi virtuali resterebbero quelli anche da processo a processo, i suoi indirizzi fisici cambierebbero e così i processi che non sono in esecuzione quando avviene il rimpiazzamento e il nuovo caricamento in memoria del buffer non avrebbero gli indirizzi fisici aggiornati nella loro tabella di corrispondenza, e questo effettivamente è inaccettabile.

Il problema a cui lei si riferisce, però, esiste indipendentemente dai driver, per ogni pagina condivisa. Se una pagina condivisa viene rimpiazzata e, successivamente, riportata in memoria fisica, andrà ad occupare quasi certamente un indirizzo fisico diverso. Chiaramente, tutti i processi che condividono la pagina dovranno avere il loro indirizzo virtuale della pagina in questione rimappato sul nuovo indirizzo fisico. Quindi, un meccanismo che risolva questo problema deve essere previsto in ogni caso, in presenza di pagine condivise. Il problema a cui si riferisce il professore, invece, è che la routine di page fault deve svolgere operazioni di I/O (caricamento e possibilmente salvataggio di pagine), quindi richiede le interruzioni abilitate. Quindi, se un driver causasse page-fault (perché il buffer di I/O era stato tolto dalla memoria fisica), gli interrupt verrebbero riabilitati dalla routine di page-fault, e l'esecuzione del driver non sarebbe più atomica.

Perché la `c_driver_io` non può richiamare direttamente la `sem_signal`()? Sul libro ciò viene giustificato dicendo che il driver non è un processo e non ha quindi un descrittore. Non è corretto che la `sem_signal`() operi sul descrittore del processo che si trova in esecuzione quando è partito il driver?

No, non è corretto. Il processo che era in esecuzione (chiamiamolo P_1) quando è partito il driver non ha, in generale, alcun legame con l'operazione di I/O che il driver sta servendo (il driver la sta eseguendo per conto di un altro processo, P_2, che in questo momento è bloccato). Il processo P_1 deve ripartire da dove lui era stato interrotto dal driver, non da dove il driver si trova quando invoca la `sem_signal`. Il driver non ha né uno stack proprio, né un proprio descrittore di processo. Se vogliamo, lo stack lo *chiede in prestito* al processo P_2, con la promessa di farglielo ritrovare così come lo aveva lasciato. Questo è possibile per come è fatto lo stack: basta aggiungere nuovi elementi in cima e poi toglierli. Questo prestito non è possibile, invece, per il descrittore di processo: qui, ogni nuovo valore cancella il precedente, quindi il driver, se ci scrivesse sopra il suo stato, non potrebbe più restituire il descrittore al processo P_2 *così come lo aveva trovato*.

Pagina 84, volume IV: "*il buffer di memoria [...] deve risiedere nello spazio di indirizzamento comune (e non deve essere soggetto a rimpiazzamento)*". Perché non deve essere soggetto a rimpiazzamento?

Perché altrimenti il driver, cercando di scrivere nel buffer, potrebbe causare un page-fault. La routine di page-fault, in genere, ha bisogno di sospendere il processo che ha causato il page-fault in attesa che la pagina mancante sia caricata. Ma il driver non è (realizzato come) un processo, quindi non possiamo sospenderlo. Il discorso sarebbe diverso se al posto del driver usassimo la coppia handler/processo esterno. In quel caso il processo esterno potrebbe essere tranquillamente sospeso, se dovesse generare un page-fault nell'accesso al buffer. Anche in questo caso, però, il sistema è molto più semplice se il buffer non può essere rimpiazzato, perché gestire il rimpiazzamento di una pagina che si trova nella memoria virtuale di più di un processo (in questo caso: processo che ha fatto la richiesta e processo esterno) è molto complicato.

Nel caso di I/O con processi esterni cosa succede se metto la `sem_signal`() nel primo if che controlla l'ultimo trasferimento da compiere?

Possiamo risvegliare il processo che ha richiesto gli N byte prima di leggere l'N-esimo.

Nelle operazioni di lettura e scrittura (`read_n` e `write_n`) il buffer di memoria è posto nel modulo `utente.cpp`. Perché? Potrebbe essere in `sistema.cpp`?

No. Immagini di essere un programmatore di applicazioni per questo sistema. Nel suo programma, ad un certo punto, vuole che l'utente inserisca dei dati da tastiera. Invocherà la primitiva `read_n` passandole come numero di interfaccia il numero che (a priori) identifica la tastiera e, come buffer, un array che *lei* avrà allocato nel suo programma (è a lei che servono questi dati). Come pensa

di poterlo allocare a livello di privilegio sistema? E, quando anche lo avesse allocato, come pensa di poterne leggere il contenuto per farci qualcosa?

Nella trattazione di un'operazione di lettura abbiamo considerato sia il descrittore di I/O che la primitiva di I/O a livello sistema. Gli stessi elementi, utilizzando handler e processi esterni, sono stati collocati a livello di privilegio I/O. C'è qualche motivo particolare per cui è stata fatta questa differenza?

Da un punto di vista architetturale, è meglio collocare tutto ciò che riguarda l'I/O a un livello intermedio tra il livello sistema e il livello utente (in modo che il normale utente non possa accedere direttamente ai dispositivi, ma evitando, al tempo stesso, che bug nel sotto-sistema di I/O si propaghino nel nucleo vero e proprio). Per come è fatto il nucleo mostrato a lezione, tale soluzione non può essere adottata nello schema primitiva+driver, in quanto:

- i driver non possono invocare primitive potenzialmente bloccanti (non avendo un descrittore di processo);
- i driver, quando l'operazione di I/O termina, devono inserire in coda pronti il processo che aveva richiesto l'operazione stessa, e si era bloccato.

Per ottenere questo ultimo effetto, si usa normalmente un semaforo di sincronizzazione: il processo richiedente esegue la `sem_wait`, bloccandosi in attesa che qualcuno esegua la corrispondente `sem_signal` che lo risveglia. La `sem_signal` è però una primitiva potenzialmente bloccante, quindi il driver non può invocarla. Allora abbiamo che:

- per ottenere lo stesso effetto della signal, senza invocarla, il driver deve manipolare le code dei processi (estrarre il processo dalla coda del semaforo e inserirlo nella coda pronti);
- le code dei processi si trovano a livello di privilegio sistema.

Quindi anche il driver deve trovarsi a livello di privilegio sistema.

Tutto questo problema non si pone nello schema primitiva+handler/processo esterno: il processo esterno, essendo appunto un processo, può tranquillamente invocare la `sem_signal` sul semaforo di sincronizzazione, quindi non ha necessità di accedere direttamente alle strutture dati (code dei processi) del sistema.

Mi era parso di capire che la scelta di collocare i descrittori e le primitive di I/O a livello sistema anziché a livello I/O fosse dovuta al fatto che in generale anche i processi sistema potessero aver necessità di invocare le primitive di I/O.

Sì, il livello sistema può avere la necessita di invocare le primitive di I/O (ad esempio per inviare messaggi di errore all'operatore, o per accedere al dispositivo di swapping), ma la chiave è quel *può*. Si può infatti organizzare il sistema in modo che tale necessità sia ridotta al minimo, o eliminata del tutto, quindi non rinunciando a collocare tutto l'I/O ad un livello inferiore a quello sistema. È vero, comunque, che la maggior parte dei sistemi in commercio, per semplicità, non usano questa politica, e collocano tutto il sotto-sistema di I/O al livello di privilegio massimo.

Nelle `a_primitive` di I/O il salvataggio dei registri si riferisce solo a quelli utilizzati per l'elaborazione della `a_primitiva` stessa?

Sì. Infatti, assumiamo che il compilatore C++ provveda lui stesso a inserire il salvataggio dei registri usati dalla corrispondente `c_primitiva`.

Quando va in esecuzione un driver relativo alle operazioni di I/O, durante i trasferimenti intermedi l'unico processo che sicuramente non torna in esecuzione è quello che ha invocato l'operazione stessa. Nella figura 10.1 si evince che torna sempre in esecuzione il processo P_2 (quello che non ha invocato l'operazione). Solo perché si sta supponendo che i possibili processi sono 2? Se fossero 3 potrebbe accadere che ad un certo punto al posto di P_2 (ma non P_1, ipotesi: operazione non ancora conclusa) vada in esecuzione un processo P_3? Io ho pensato di no, in quanto per andare in esecuzione dovrebbe avvenire la situazione in cui P_3 da sbloccato abbia priorità maggiore di P_2 (preemption). Ma può accadere che si sblocchi mentre e in corso una operazione I/O e si sostituisca a P_2?

P_3 può andare in esecuzione anche perché P_2 esegue una `sem_wait` su un semaforo occupato (ad esempio, P_2 cerca di eseguire una operazione di I/O sulla stessa interfaccia su cui sta lavorando P_1, nella stessa direzione), oppure perché P_3 ha una priorità maggiore di P_2 e:

- P_2 esegue una `semi_signal` su un semaforo su cui P_3 era bloccato;
- P_3 era bloccatto su un'altra operazione di I/O che termina prima di quella lanciata da P_1;
- P_3 aveva eseguito una chiamata a `delay` ed è passato il tempo richiesto,

più, forse, altri casi che ora non mi vengono in mente.

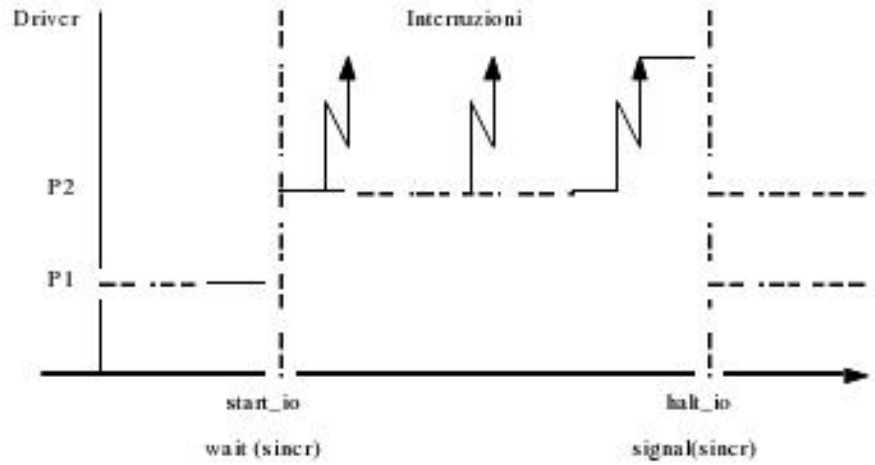


Figura 10.1: Operazione di I/O a interruzione di programma

Perché nel capitolo relativo all'I/O i sottoprogrammi di interfaccia per le primitive di semaforo sono in *sistema.s*? Nel capitolo precedente i sottoprogrammi di interfaccia erano in *utente.s*.

Perché vanno usati anche da livello sistema (da parte delle pritive di I/O e dei rispettivi processi esterni) e, da livello sistema, non è possibile chiamare un sottoprogramma che si trova a livello utente. Ne resta comunque una copia a livello utente, per gli usi normali.

Nel caso di utilizzo delle primitive di I/O con segmentazione attivata quando le `a_prim_io` ricopiano i parametri per le `c_prim_io`, devono modificare anche il campo RPL del selettore del segmento contenente il buffer in modo da eliminare il problema del cavallo di troia, giusto?

Sì, esatto.

Capitolo 11

Processi esterni

Quali sono i malfunzionamenti che si possono presentare nel caso in cui il corpo del processo esterno fosse scritto nel modo seguente? (supponiamo che faccia ingresso):

```
void esterno (int h){
        char a;
        des_io* p_desi;
        p_desi=&desinti[h];
        for(;;){
                (p_desi->cont)--;
                if ((p_desi->cont)==0) {
                        halt_input(p_desi>ind_reg.ctr_io.iCTRI);
                        sem_signal(p_desi->sincr);
                }
                input_b(p_desi->ind_reg.in_out.iRBR,a);
                *(p_desi->punt)=a;
                p_desi->punt++;
                wfi();
        }
}
```

è possibile che riportare il processo che ha comandato l'operazione di I/O dalla coda gestita dal semaforo di sincronizzazione a quella dei pronti (eventualmente nella coda `esecuzione` se c'è preemption) sia una conferma, per gli altri processi, che il buffer utilizzato per l'ingresso dati sia in uno stato consistente e che quindi possono accedervi per leggere i dati in esso contenuto?

Per gli *altri processi* non lo so, ma sicuramente è un segnale per il processo che aveva richiesto l'operazione di I/O e che viene risvegliato dalla `sem_signal(p_desi->sincr)`. Il processo potrebbe andare in esecuzione e utilizzare il buffer, anche se l'ultimo dato non è stato ancora trasferito.

Vorrei un chiarimento riguardo ai processi esterni: perché devono avere struttura ciclica? Se togliessi il ciclo infinito dal codice del processo esterno, cosa succederebbe?

Quando il processo esterno arriva alla `wfi`(), viene salvato il suo stato in quel punto. La prossima volta che il processo esterno verrà messo in esecuzione, ripartirà da quel punto e, in assenza del ciclo, terminerà per sempre.

Per quanto riguarda lo stato delle pile nella chiamata di un processo esterno, al termine di un processo esterno, a causa della chiamata di una primitiva `wfi`(), va in esecuzione un sottoprogramma d'interfaccia `_wfi` che salva l'indirizzo di ritorno in pila I/O: per caso tale indirizzo si riferisce all'istruzione `RET` in `io.s`?

Certo, è semplicemente l'effetto dell'esecuzione dell'istruzione `INT`.

Al termine dell'ultima operazione di I/O in un processo esterno, dopo aver letto/scritto l'ultimo dato, viene eseguita una signal sul semaforo di sincronizzazione. Qualora questa preveda preemption (e non essendo specificato diversamente sul libro, presumo che sia così), è possibile che venga messo in `esecuzione` il processo precedentemente bloccato dalla `sem_wait`(`sincr`) (il processo che aveva richiesto i dati in I/O), e quindi il processo esterno non arriverebbe ad invocare la `wfi`() per l'ultima volta. In questo caso, non verrebbe inviato l'ultimo *EOI* all'interfaccia e, qualora in un secondo momento venisse richiamato lo stesso processo esterno, questo non riprenderebbe dall'istruzione corretta.

È vero che la `sem_signal`(`sincr`) potrebbe causare la preemption del processo esterno da parte del processo che aveva richiesto l'I/O (in realtà si assume che i processi esterni abbiano priorità maggiore di tutti gli altri processi, ma possiamo anche fare a meno di questa assunzione). Ciò non causa comunque problemi, perché, per effetto della preemption, il processo esterno verrà inserito in coda pronti e, quindi, prima o poi verrà rischedulato e potrà eseguire la `wfi`. Inoltre, fino a quando il processo esterno non esegue la `wfi` (inviando quindi la parola *EOI* al controllore), il controllore delle interruzioni non farà passare richieste di interruzioni a priorità minore o uguale a quella gestita da quel processo esterno. In particolare, altre richieste da parte della stessa interfaccia (associata a quel processo esterno) dovranno aspettare. Quindi, non potrà mai andare in esecuzione l'`handler` di quel processo esterno, che è l'unico che può rimetterlo forzatamente in esecuzione.

Nel momento in cui viene invocata la `activate_pe`() l'indirizzo della prima locazione di memoria che contiene il corpo del processo, che si trova nello spazio di indirizzamento comune, viene immesso in testa alla pila sistema del processo esterno in modo tale da essere caricato in *EIP* quando si esegue la `IRET` che si trova nel corrispondente handler oppure viene ricopiato nella sezione text del processo stesso?

La prima. Comunque, per essere precisi, non stiamo parlando della *prima locazione di memoria che contiene il corpo del processo*, ma dell'entry-point del

corpo del processo, cioè la prima istruzione da eseguire (che non necessariamente sarà quella all'indirizzo più basso)

Pagina 104, volume IV: a quanto ho capito l'ultima parte è relativa all'ipotesi che, anche se durante l'esecuzione di un processo esterno, si potesse verificare la riesecuzione dello stesso processo esterno, non ci sarebbero problemi in quanto il primo processo esterno riprenderebbe sempre e comunque dalla prima istruzione del ciclo for recuperando le azioni che sarebbero dovute essere eseguite da una seconda istanza. Ho capito bene? E tutto ciò è dovuto alla `salva_stato` chiamata dalla `wfi`? Capito questo per quanto mi sforzi la parte *l'invio della parola EOI [...] prima che il processo esterno stesso si sia bloccato* rimane avvolto nel mistero.

L'invio della parola *EOI* comunica al controllore d'interruzione che la sua richiesta di interruzione, chiamiamola X, è stata servita. Questo autorizza il controllore di interruzione a lasciar passare richieste di interruzione (se ve ne sono) anche a priorità minore o uguale a X (quando X era ancora sotto servizio, il controllore di interruzione memorizzava le richieste a priorità minore o uguale, ma non le faceva passare). In particolare, potrebbe esserci una richiesta pendente dalla stessa interfaccia associata a X, che quindi, dopo l'invio di *EOI*, potrebbe ora passare e arrivare al processore. Nella soluzione corretta, l'invio di *EOI* avviene all'interno di `wfi`, che è eseguita a interruzioni disabilitate e, prima di tornare, blocca il processo esterno (chiama lo schedulatore): quindi la nuova richiesta può essere servita dal processore solo dopo che il processo esterno si è bloccato *al posto giusto*, e quindi può essere quindi fatto ripartire normalmente. Se, però, inviassimo *EOI fuori* dalla `wfi`, ecco che si potrebbe presentare il problema descritto a pagina 104: il processo esterno potrebbe essere bloccato da una nuova istanza di se stesso (subito dopo l'invio di *EOI* e prima di entrare dentro `wfi`).

Pagina 106, volume IV: "*pertanto, il valore di p_desi di quel processo non cambia se un altro processo esegue un'operazione di lettura su un'altra interfaccia, prima che sia conclusa l'operazione in esame*". Non capisco.

Anche se più processi esterni eseguono lo stesso corpo (quindi lo stesso codice contenuto a pagina 106) contemporaneamente, ognuno ha la sua pila distinta, quindi ognuno ha la sua copia privata delle variabili locali (che, come sappiamo, vengono allocate sulla pila), come `p_desi`.

Pagina 102, volume IV: "*una interruzione a precedenza maggiore di quella sotto servizio modifica il descrittore del processo in esecuzione: da qui discende la necessità che il processo esterno si blocchi eseguendo la primitiva* `wfi`() *che salva il suo stato*". Ho capito la necessità della `wfi`() (per salvare lo stato, inviare *EOI* e chiamare schedulatore) ma quello che non capisco è la correlazione definita dalle parole *da qui discende*. Un'interruzione a precedenza maggiore cosa dovrebbe modificare del descrittore del processo esterno, al massimo farebbe una semplice `salva_stato` e perché ciò sarebbe negativo?

Il fatto è questo: se il processo non salvasse il suo stato alla fine, lo stato salvato nel suo descrittore resterebbe quello salvato a causa dell'interruzione a privilegio superiore (che avrà, ad esempio, *EIP* che punta ad un punto intermedio del corpo programma eseguito dal processo). Quindi, la prossima volta che il processo esterno venisse messo forzatamente in esecuzione, non ripartirebbe dall'inizio, ma dal quel punto causale. Se, invece, il processo esterno non potesse essere interrotto, potremmo eliminare il ciclo `for` (dal corpo del processo) e la chiamata alla routine `salva_stato` (dalla primitiva `wfi`): ad ogni attivazione, il processo partirebbe comunque dall'inizio, perché il suo descrittore di processo resterebbe sempre costante (in questo schema, se vogliamo eseguire comunque una sola volta la parte di corpo che, nell'altro schema precede il `for`, dovremmo prevedere un flag nel descrittore di I/O associato al processo...)

Inoltre un'interruzione a precedenza maggiore viene comunque servita prima della chiamata alla `wfi`(), quindi la presnza della `wfi`() non impedisce ciò.

Infatti non vogliamo impedire che il processo esterno venga interrotto dalle interruzioni a privilegio maggiore. Anzi, è proprio per permettere queste interruzioni che siamo passati dal driver alla coppia handler/processo esterno.

Nel corpo di un processo esterno, può accadere che durante l'esecuzione della `sem_signal`() ci sia una preemption e venga messo in esecuzione un processo diverso dal processo esterno stesso? Se ciò potesse effettivamente avvenire, il processo esterno, una volta riattivato, ripartirebbe dal punto successivo alla `sem_signal` per poi bloccarsi con la `wfi`()?

Sì. Il processo esterno, da questo punto di vista, è un processo come tutti gli altri.

Perché quando viene sospeso un processo esterno, il suo puntatore non viene messo nella coda dei processi pronti? Forse perché è sempre presente nella *coda dei processi esterni bloccati in attesa di una particolare interruzione*?

Esattamente.

Perché la *coda dei processi bloccati in attesa di un particolare tipo di interruzione* è costituita da un solo elemento?

La coda a cui si fa riferimento in quel passaggio è quella da cui l'handler, associato a quel tipo di interruzione, estrarrà il descrittore del processo esterno da mandare in esecuzione. La possiamo pensare come una *coda* per uniformità, ma il suo scopo è solo quello di mantenere il puntatore al processo esterno associato a quel tipo di interruzione.

Nel meccanismo dei processi esterni c'è un seppur minimo spreco di memoria, poiché un processo esterno arrivato alla `wfi`() si blocca e rimane in attesa di essere sbloccato da un'interruzione. Se tale attesa diventa infinita, la memoria che occupa nella coda dei bloccati è effettivamente memoria sprecata. Mi chiedevo come ci si comportasse nel caso di architetture hardware estremamente memory-limited, dove però sia necessario far ricorso ai processi esterni, come i sistemi embedded.

Capisco la sua curiosità, ma non ho una risposta precisa per la sua domanda, perché i sistemi embedded sono tanti e diversi, e le soluzioni adottate sono (per definizione) specifiche per quel problema. Diciamo che, per quanto riguarda questo caso particolare che lei pone, non conosco un sistema che abbia abbastanza memoria per poterci costruire un sistema multiprogrammato, ma non abbastanza per contenere tutti i descrittori di processo richiesti, quindi non so se qualcuno ha mai dovuto risolvere il problema che lei pone. Io, di fronte a questo caso, proverei ad accorpare più processi esterni in uno, se i vincoli del problema lo permettono.

Nel libro viene detto che la `activate_pe` ha il seguente prototipo:

```
 void activate_pe(void f(int),int a,int prio,char
liv,short &identifier,proc_elem*& p,bool& risu);
```

Ma chi può richiamare questa primitiva? Il `main` non può: non conosce la funzione da passare (definita nei moduli I/O) né il tipo `proc_elem`. I moduli I/O neanche: non conoscono `proc_elem`. I moduli sistema nemmeno: non conoscono la funzione da passare...

Sì, è vero. Il problema è `proc_elem`, che non è conosciuto da nessuno se non dai moduli di sistema. Nel nucleo effettivamente realizzato, è stato sostituito da una variabile di tipo enumerazione (assumendo di assegnare a priori, ad ogni interfaccia, una costante che la identifica). In ogni caso, la parte di inizializzazione (non vista a lezione) contravviene alla regola del *non collegamento* tra i vari moduli: il modulo di sistema inizializza le strutture dati di sistema, quindi invoca la routine di inizializzazione del modulo di I/O (quindi, il modulo di sistema deve conoscere l'indirizzo di questa routine). È questa routine che chiama la `activate_pe` per ogni processo esterno necessario.

Nel corpo di un processo esterno per operazioni di ingresso o uscita cosa succede se sposto `halt_input`(), presente nel primo `if`, nel secondo?

Il processo esterno vuole eseguire la `halt_input` perché è arrivato all'ultimo byte da leggere (degli N richiesti), e quindi bisogna disabilitare l'interfaccia a generare ulteriori interruzioni (fino a quando un qualche processo non eseguirà un'altra operazione di lettura sulla stessa interfaccia). Se, però, spostiamo la `halt_input` dopo la `inputb`, l'iterfaccia può fare in tempo a generare un'altra interruzione (infatti, la `inputb`, poiché legge dal buffer di ingresso dell'interfaccia, autorizza l'interfaccia a generare una nuova interruzione, se ha un nuovo dato disponibile). Questo interrupt in più farà attivare il processo esterno associato all'interfaccia una volta in più (N+1 rispetto alle N richieste). Il nuovo dato verrà letto e scritto, in memoria, dopo l'N-esimo e, quindi, molto probabilmente fuori dal buffer predisposto per la lettura, andando a corrompere il valore di qualche altra variabile.

Chi lo stabilisce il legame tra processo esterno ed handler? La `activate_pe`?

Sì. Gli handler esistono già dall'avvio del sistema, uno per interfaccia, e ognuno è scritto per mettere in esecuzione quel processo che si trova nella coda associata alla propria interfaccia. All'inizio, però, queste code sono tutte vuote. È la `activate_pe` che le riempie.

Nel libro si parla, in relazione ai processi esterni, di code dei processi bloccati. Queste code sono indirizzate con delle variabili puntatore a livello sistema come la `pronti` o `esecuzione`?

Sì. Tenga comunque presente che, per ogni interfaccia, si tratta di una coda di un solo elemento (in pratica un puntatore ad un unico descrittore di processo: il descrittore del processo esterno associato all'interfaccia). Viene trattata come coda per uniformità.

Esattamente cosa fa la `activate_pe`?

Crea un nuovo descrittore di processo (che sarà il descrittore del processo esterno), associandovi una pila sistema, una pila utente e il corpo del processo (passato come argomento alla `activate_pe` stessa), e lo inizializza (tutte queste funzioni, nell'implementazione disponibile del nucleo, sono svolte dalla procedura ausiliaria `crea_proc`). Quindi, inserisce il descrittore di processo appena creato nella coda dei processi associata alla interfaccia richiesta (nel libro, la `activate_pe` richiede come parametro direttamente la coda di processi in questione, mentre nell'implementazione l'interfaccia viene specificata tramite un enumerato).

Il parametro `h` che compare come parametro formale nel processo esterno chi lo inserisce?

La `activate_pe`. Lo inserisce nella pila utente, lì dove il corpo del processo (essendo ottenuto da una traduzione dal C++) si aspetta di trovarlo.

Il richiamo dello schedulatore nella `wfi` può portare alla selezione di: processo interrotto nelle esececuzioni intermedie (ok), processo divenuto pronto nell'esecuzione finale (ok), oppure processo interrotto (che processo? interrotto quando e da cosa?)

La rilegga così:

- nelle esecuzioni intermedie, seleziona il processo interrotto;
- nell'esecuzione finale, seleziona il processo interrotto o quello divenuto pronto.

Capitolo 12

I/O in DMA

Nel caso di utilizzo del controllore dma non mi sembra sia necessario che il buffer interessato nell'operazione di I/O debba risiedere nello spazio di indirizzamento comune, infatti l'indirizzo fisico del buffer viene caricato nel *MAR* e là rimane a prescindere dal processo in esecuzione. Inoltre il processo esterno che va in esecuzione a fine operazione si limita a disabilitare l'interfaccia a generare richieste di interruzione e a eseguire la signal sul semaforo di sincronizzazione. Dunque perché vi è scritto che deve essere nello spazio di indirizzamento comune?

Perché concettualmente, il controllore DMA trasferisce i dati mentre il processo che li ha richiesti non è in esecuzione (è bloccato sul semaforo di sincronizzazione). In generale, dobbiamo pensare che, mentre un processo non è in esecuzione, il suo spazio di indirizzamento privato non è accessibile. Ciò può non essere vero in qualche implementazione (tipicamente, tutte quelle con memoria virtuale paginata), ma può essere vero in altre.

Perché nelle operazioni di lettura e scrittura in DMA il buffer deve risiedere nello spazio di indirizzamento comune?

In realtà non è necessario.

Forse perché (essendo il buffer luogo in cui vengono immessi o prelevati i byte) deve essere sempre accessibile dall'utente, mentre se fosse contenuto nel modulo *sistema* non sarebbe più possibile per l'utente accedervi per immettere o prelevare byte?

No, questo non c'entra niente. Il termine *spazio di indirizzamento comune* indica che è comune a tutti i processi, non che è dichiarato nel modulo *utente* piuttosto che nel modulo *sistema*. Nel modello visto a lezione, lo spazio di indirizzamento di ogni processo ha due porzioni che sono a comune con tutti gli altri processi: una a livello utente (lo spazio in cui risiedono, ad esempio, le variabili globali, cioè quelle dichiarate fuori dal corpo di ogni funzione) e uno a livello sistema (dove risiedono le strutture dati globali di sistema, come le code dei processi). Nel testo si intende che il buffer, il cui indirizzo viene passato alla `dmaleggi` e `dmascrivi`, (buffer che, ovviamente, deve trovarsi a livello utente) deve essere

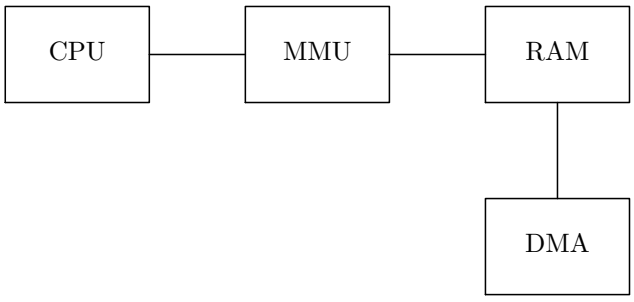
allocato nel primo di questi spazi comuni. Infatti, nel codice di esempio, viene dichiarato fuori dal corpo del processo.

Il controllore DMA vuole nel suo registro *MAR* l'indirizzo fisico della prima locazione di memoria da o verso cui deve effettuare il trasferimento. Dato che la consecutività in memoria fisica si mantiene solo all'interno della pagina, se pongo in *MAR* un indirizzo prossimo alla fine della pagina (immaginiamo con i 12 bit meno significativi uguali a 1), il controllore prosegue in memoria fisica e opera nelle locazioni successive alla pagina, dove non sappiamo cosa ci possa essere. C'è qualcosa che controlla questo problema, eventualmente "spezzando" in due parti il trasferimento?

Sì e No. Nel sistema che abbiamo studiato a lezione (versione semplificata di un PC IBM ormai vecchiotto), non c'è un meccanismo hardware che aiuti a risolvere questo problema. In presenza di memoria virtuale, il software deve provvedere a ordinare trasferimenti esclusivamente all'interno di una pagina, o a spezzarli in tanti trasferimenti. Nei PC più moderni, invece, le interfacce che sono in grado di accedere direttamente alla memoria prevedono, in genere, non un unico registro *MAR*, ma una lista o un vettore di descrittori di trasferimenti (dove ogni descrittore contiene un indirizzo e un contatore). La lista (o il vettore) devono essere allocati in memoria, e l'indirizzo del primo elemento deve essere scritto in un registro dell'interfaccia. L'interfaccia legge quindi i vari descrittori, eseguendo autonomamente i diversi trasferimenti previsti. Questo è un meccanismo (detto dello scatter/gather) utile in diverse occasioni, tra cui quella a cui lei fa riferimento.

Perché il buffer usato in un trasferimento dma non deve essere soggetto a rimpiazzamento?

Non si può usare il meccanismo del rimpiazzamento sui buffer usati in trasferimenti DMA, perchè il DMA bypassa anche la MMU e tutta la gestione della memoria. Lo schema, tralasciando la cache ed evidenziando il ruolo di MMU, è:



Pensi a cosa accadrebbe nel seguente scenario:

1. il processo P_1 ordina una `dmascrivi` nel buffer all'indirizzo virtuale v_1 (supponiamo, per semplicità, che il buffer corrisponda ad una intera pagina virtuale, V_1);

2. `dmascrivi` programma il controllore DMA in modo che scriva in RAM, nella pagina fisica F_1 corrispondente alla pagina virtuale V_1;

3. P_1 si blocca, in attesa che il trasferimento DMA sia completato. Viene schedulato P_2;

4. P_2 accede ad una pagina virtuale V_2, che non è presente in memoria fisica;

5. va in esecuzione la routine di rimpiazzamento, che sceglie proprio la pagina fisica F_1 come vittima del rimpiazzamento: copia F_1 nello spazio di swap, modifica la tabella delle pagine di P_1 in modo che V_1 risulti assente, copia il contenuto di V_2 dallo spazio di swap nella pagina fisica F_1 e modifica la tabella delle pagine di P_2 in modo che V_2 risulti presente;

6. il controllore DMA è completamente all'oscuro di tutto ciò che è successo al punto precedente, e continua a trasferire i dati dall'interfaccia nella pagina fisica F_1, distruggendo il contenuto della pagina V_2 dell'incolpevole processo P_2.

Capitolo 13

Segmentazione

Nella struttura dei segmenti *TSS*, che fungono da descrittori di processo in presenza di segmentazione, come rappresentata sul libro, non è previsto un campo per il contenuto di *CR3*.

Credo che la mancanza sia semplicemente dovuta al fatto che il professore ha organizzato il discorso paginazione/segmentazione in modo che si potesse parlare dell'una indipendetemente dall'altra (a scopo didattico). Nel *vero TSS*, il campo *CR3* c'è.

Non ho capito come funziona il modello di memoria flat...

Il modello flat serve ad approssimare un modello in cui la segmentazione non esiste, anche in un sistema in cui la segmentazione non può essere disabilitata in alcun modo (come nei processori Intel x86, in cui protezione e segmentazione sono strettamente intrecciate). In pratica, si creano solo segmenti con base 0 e limite 0xFFFFFFFF. Tali segmenti occupano tutta la memoria lineare del processo, e sono completamente sovrapposti l'uno sull'altro. In pratica, è quasi come se non ci fossero (è semplice vederlo: la traduzione da indirizzo logico a lineare non ha alcun effetto). Dico *quasi* perché gli effetti legati alla protezione continuano ad essere presenti: in *CS* deve essere caricato un selettore di segmento codice, in *SS* un segmento dati scrivibile e in *DS* (e negli altri) un segmento dati. Quindi, sono necessari almeno due (descrittori di) segmenti: un segmento codice e un segmento dati scrivibile (che va bene sia per *SS* che per *DS*). Inoltre, il livello di privilegio del processore continua a essere dato dai due bit meno significativi di *CS* e, in ultima analisi, dipende dal campo *DPL* del segmento codice corrente. Quindi, se vogliamo avere due livelli di privilegio (per realizzare un nucleo protetto) dobbiamo prevedere anche un altro segmento codice, con *DPL* sistema. Questo ci porta, infine, ad avere anche un altro segmento dati scrivibile, sempre con *DPL* sistema, altrimenti non potremmo caricarlo in *SS* quando il livello di privilegio del processore è passato a sistema.

...in particolare: perché il codice, i dati e la pila devono essere allocati in zone diverse del rispettivo segmento?

Questi quattro descrittori di segmento, come detto prima, devono avere stessa base (0) e stesso limite (0xFFFFFFFF) e sono, quindi, sovrapposti in memoria

lineare. Se allocassimo, per esempio, il codice (nel segmento codice utente) e la pila (nel segmento dati utente) agli stessi offset nei rispettivi segmenti, codice e pila finirebbero agli stessi indirizzi in memoria lineare.

Da quello che ho capito, la frammentazione della memoria lineare è dovuta al fatto che i segmenti hanno lunghezze diverse e che il rimpiazzamento coinvolge segmenti interi, quindi può accadere che, per caricare in memoria lineare un segmento, se ne debbano rimpiazzare uno o più altri, ma non è detto che il nuovo segmento occupi, per intero, lo spazio liberato (in questo caso resta dello spazio inutilizzato, un *buco*. Questo può far sì che, in memoria lineare, ci sia dello spazio libero sufficiente ad accogliere altri segmenti, ma che tale spazio sia suddiviso in più *buchi*, non contigui tra loro). Nel caso in cui sia attiva anche la paginazione, la frammentazione non si presenta solo se la memoria logica risiede tutta in memoria lineare (non è una memoria virtuale, quindi non è necessario effettuare rimpiazzamenti di segmenti), ma in caso contrario può verificarsi, giusto? (i *buchi* sono costituiti da intervalli di indirizzi lineari (virtuali)).

Esattamente. C'è da dire, però, che nel caso segmentazione+paginazione, anche quando la memoria logica è più grande della memoria lineare, la frammentazione (in memoria lineare) crea meno problemi di prestazioni, in quanto *spostare un segmento* significa solo modificare alcune tabelle delle pagine (in modo che i nuovi indirizzi lineari puntino alle vecchie pagine fisiche o alle vecchie pagine in memoria di massa).

Il fatto che nei programmi scritti usando l'ambiente GNU si usi un indirizzamento lineare, c'entra qualcosa con l'utilizzo del modello flat? Cioè: nella traduzione vengono prodotti segmenti sovrapposti in memoria lineare, che fanno apparire al programmatore la memoria non segmentata e poi ci pensa l'ambiente a posizionare le varie entità ad offset differenti nei rispettivi segmenti(o qualcosa del genere)?

Sì. L'*ambiente*, in questo caso, è semplicemente il collegatore.

Le routine che gestiscono la paginazione (se questa è attiva in presenza di segmentazione), come fanno ad accedere alle tabelle delle pagine? Anche queste devono essere mappate nello spazio logico (cioè devono esserci dei segmenti accedendo ai quali si accede alle tabelle)?

Sì.

Suppongo che le routine usino indirizzamento segmentato.

In un ambiente propriamente segmentato, tutti gli indirizzi usati dal programmatore sono logici. Quindi, anche gli indirizzi usati dalle routine che gestiscono la paginazione.

Mi pare di aver capito che le pagine che costituiscono un segmento, non hanno necessariamente lo stesso livello di privilegio del segmento. Non riesco a capire a cosa possa servire avere, all'interno di un segmento, pagine di livello di privilegio differente da quello del segmento. L'unica cosa che mi viene in mente è che, in questo modo, è possibile far sì che, per l'accesso alle pagine del segmento, siano considerate le sole regole imposte dalla paginazione. Quindi magari, un unico segmento dati, di livello di privilegio utente (anziché usare un segmento per ogni livello di privilegio, se ne usa uno solo), costituito sia pagine di livello utente che da pagine di livello sistema, sarebbe sempre utilizzabile dal processore e garantirebbe la protezione delle pagine sistema. In conclusione si può risparmiare sul numero di segmenti? È forse questo uno degli scopi?

In un certo senso. Il motivo principale è quello di permettere di realizzare la protezione nel modello flat (in questo modello, essendo i segmenti tutti sovrapposti, la protezione garantita a livello di segmenti non ha alcun senso).

Nel caso si usi il modello di memoria flat, la tabella LDT non si usa? Da quello che ho capito mi pare che non serva, o che, comunque, se ne possa fare a meno, giusto? La *LDT*, può essere usata per rendere privata una parte dello spazio di indirizzamento di un processo (includendo in essa un sottoinsieme dei selettori e dei descrittori dei segmenti che lo compongono). Tali segmenti sono accessibili soltanto dai processi che usano quella *LDT*. Nel caso di memoria flat, invece, dato che tutti i segmenti sono sovrapposti, non ha senso (credo) parlare di segmenti privati di un singolo processo (non c'è bisogno di utilizzare una particolare *LDT* per conoscere base e limite di un segmento, dato che tali informazioni sono le stesse per tutti). In questo caso i descrittori, contenuti nella *GDT* (che devono essere almeno due per ogni livello di privilegio che si intende usare) vengono utilizzati, da tutti i processi, solo per cambiare livello di privilegio e per immettere in *SS* e *DS* descrittori di livello appropriato. La suddivisione dello spazio di indirizzamento in condiviso e privato, così come, la regolamentazione dell'accesso alle pagine contenute in ogni segmento (in questo caso, ogni segmento contiene anche pagine relative ad altri segmenti e quindi, possibilmente, di livello di privilegio differente) è delegata al meccanismo di paginazione. Quindi mi pare che si possa fare a meno della *LDT* (nel nucleo realizzato da voi, infatti, mi sembra che non sia usata).

È tutto come dice lei. Solo, le consiglio di pensare al modello flat in modo molto più intuitivo: è un modo per simulare un sistema in cui esiste soltanto la paginazione, all'interno di un sistema che è, invece, segmentato e paginato. I segmenti non giocano alcun ruolo all'interno del modello.

Cosa significa in termini spiccioli che un segmento in memoria lineare è allineato alla pagina?

Che il suo indirizzo di base deve essere un multiplo della dimensione di una pagina. In altre parole, che il primo byte del segmento deve coincidere col

primo byte di una qualche pagina.

Per quale motivo in tal caso deve avere i 12 bit più significativi uguali a 0 ?

Non i più significativi, ma i *meno* significativi. Affinché un numero x sia multiplo di y, il resto della divisione di x per y deve essere 0. Nel nostro caso $y = 4096 = 2^{12}$ e il resto di x diviso y coincide con i 12 bit meno significativi di x.

Se si utilizza un modello flat si considera la memoria come un unico segmento da 4GiB (base 0x00000000, limite 0xFFFFF e granularità di pagina)? ...

Intanto una precisazione: nel modello di segmentazione visto nel corso, non è possibile avere un unico segmento: se questo segmento fosse di tipo codice, potremmo caricare il suo selettore in *CS*, ma non potremmo usarlo come pila, in quanto il processore permette di caricare in *SS* solo segmenti dati scrivibili; se, al contrario, questo unico segmento fosse di tipo dati (scrivibile), potremmo caricare il suo selettore in *SS*, *DS*, etc. ma mai in *CS*. Quindi, sono necessari almeno due segmenti: codice e dati (scrivibile). Nel modello flat, questi due segmenti sono sovrapposti: entrambi hanno base 0x00000000 e limite 0xFFFFF, con granularità di pagina (di fatto by-passando la protezione dovuta al tipo dei segmenti).

... Questo unico segmento deve contenere solo pila, dati e codice del processo oppure gli indirizzi a partire da 0 fino a 2^{n-1} sono occupati dalla memoria fisica?

Se scegliamo di mappare l'intera memoria fisica nella memoria lineare di ogni processo, poiché entrambi quei segmenti abbracciano l'intera memoria lineare, automaticamente la memoria fisica sarà visibile all'interno di quei segmenti, agli indirizzi in cui l'abbiamo mappata in memoria lineare (ad esempio, da 0 a 2^{n-1})

Selettore di segmento *TSS* = *ID* del processo?

Sì, nel senso che l'*ID* di cui si parla nella prima parte del libro di calcolatori altro non è che il selettore di segmento *TSS*.

C'è un controllo di qualche tipo in modo tale che non si possa mai verificare una sovrapposizione non voluta dei segmenti? Se non esiste non è potenzialmente pericoloso per la stabilità del sistema?

Se si riferisce ad un controllo hardware, la risposta è no, non esiste alcun controllo di questo tipo. È un pericolo per la stabilità del sistema? Mah, anche scrivere una primitiva di nucleo che non esegue controlli su i propri parametri di ingresso lo è. Come ho scritto, è la routine di trasferimento che carica i segmenti in memoria lineare (quella va in esecuzione per effetto di un segment-fault) che deve scegliere la base del segmento da caricare, facendo in modo che non si sovrapponga con quelli già presenti. Se chi scrive questa routine sbaglia, ovviamente potrebbe accadere (tra le infinite cose che potrebbero accadere) che

alcuni segmenti possano risultare sovrapposti involontariamente. È il software, babe.

La base del *TSS* non è puntata da *EAX* al momento dell'istruzione 0?

Non la base del *TSS*, *EAX* contiene la base del *descrittore* di segmento *TSS*. La base del *TSS* è dentro il descrittore.

Diciamo che un segmento non è presente in memoria lineare (quindi è solo in memoria di massa che virtualizza la logica), quando viene chiamata la segment-fault questa si limita a modificare la base del segmento (e direttorio e tabella pagine), ma perché? Cosa mi cambia, basta questo a portarlo in memoria lineare? ...

Certo. Qualcosa è in memoria lineare se è *raggiungibile* tramite una tabella delle pagine (o tramite le strutture dati associate alla memoria lineare). Supponiamo che il segmento mancante sia grande X pagine. La routine di segment-fault deve scegliere, nella memoria lineare, un range di X pagine (consecutive) non ancora mappate, ed aggiustare le strutture dati in modo che:

- la base del segmento sia l'indirizzo della prima di queste pagine (basta scriverlo nel descrittore di segmento);
- ognuna delle X pagine sia associata al corrispondente *pezzo* di segmento nella memoria secondaria (questo possiamo scriverlo nei descrittori di pagina di quelle pagine, se abbiamo adottato l'ottimizzazione secondo cui i descrittori di pagina con P=0 contengono la posizione in memoria secondaria della pagina mancante).

... non potrei avere direttamente il page-fault come quando metto tutti i segmenti in memoria lineare (limitando a 4GiB)?

Poi ci sarà anche il page-fault, ma se non aggiustiamo le tabelle come potrebbe fare la routine di page-fault a sapere cosa è associato a quell'indirizzo lineare che ha causato il fault? Il segment-fault e il page-fault avvengono in momenti diversi. Prendiamo questo pezzo di codice:

```
movw $10, %es          # questo causerà un segment-fault
                       # se il segmento 10 non è in memoria lineare
movl %es:1000, %eax    # questo causerà un page-fault all'indirizzo
                       # linerare: (base del segmento 10)+1000
```

Vorrei sapere come funziona l'indirizzamento nel modo reale: la tabella di corrispondenza come è fatta?

Quando il processore studiato a calcolatori (che è una semplificazione dei processori Intel dal 386 in poi) lavora in *modo reale*, si comporta praticamente come il processore studiato a reti logiche (che è una semplificazione dei processori Intel precedenti al 286). A reti logiche avevate visto una architettura segmentata, in cui ogni indirizzo è composto da due componenti: *selettore* e *offset*. L'indirizzo fisico si calcola a partire da queste due componenti con la formula

selettore $\cdot 16 +$ *offset*, e ogni segmento può essere grande, al massimo, 64KiB. A calcolatori, la formula possiamo scriverla (in pseudo-C): `GDT`[*selettore*].`base` + *offset* (oppure *LDT* etc.), con la dimensione del segmento data dal campo `GDT`[*selettore*].`limite`, opportunamente esteso a 32 bit in base al campo `GDT`[*selettore*].`granularita`. Quando il processore di calcolatori lavora in modo reale, la formula resta la stessa, ma il risultato è uguale a quello di reti logiche. Per ottenere questo effetto, basta far sì che:

- `GDT`[*selettore*].`base` = *selettore* $\cdot 16$;
- `GDT`[*selettore*].`limite` = `0xFFFF`;
- `GDT`[*selettore*].`granularita` = `BYTE`

(poi, a voler essere precisi, i valori per la base e il limite, nel modo reale, non sono scritti nella tabella *GDT*, ma direttamente nei campi nascosti dei registri selettori).

Nel paragrafo relativo al trasferimento di segmenti si parla di *memoria lineare* (io per memoria lineare intendo quella virtuale, ottenuta con la paginazione e non quella fisica) ma alla fine del paragrafo mi trovo uno schema che secondo me è ambiguo perché trovo scritto *memoria fisica* al posto di memoria lineare. In questo paragrafo parliamo di segmentazione con paginazione disattivata per cui per memoria lineare intendiamo la memoria fisica?

Sì

La frammentazione della memoria si ha solo in caso di paginazione disattivata, giusto? Perché?

Sì. Cerchiamo di avere ben chiara innanzitutto una cosa: se è attiva solo la segmentazione, ogni segmento da utilizzare va caricato *per intero* e a indirizzi *contigui* (cioè in un intervallo continuo) di indirizzi lineari. Questo perché l'unico controllo che la MMU fa è che, nell'indirizzo logico, l'offset (spiazzamento) sia minore o uguale al limite, e poi somma semplicemente l'offset alla base del segmento, per ottenere l'indirizzo lineare. Quindi: possiamo liberamente scegliere l'indirizzo base di un segmento, ma poi *tutto* il segmento deve seguire quell'indirizzo. È come se i segmenti fossero dei corpi rigidi, che si possono caricare, spostare o rimuovere solo muovendoli rigidamente.

Ora, siccome i segmenti sono in genere di dimensioni diverse e hanno tempi di vita diversi (dovendo contenere il codice, i dati e lo stack di processi diversi), è chiaro che dopo un po' di tempo passato a caricare e scaricare segmenti in ordine sparso, si verrà a creare frammentazione: lo spazio disponibile tenderà a sparpagliarsi tra i vari segmenti caricati, piuttosto che accumularsi da qualche parte (per semplici motivi statistici, credo intuitivi). Andando avanti (se non si ricompatta mai lo spazio disponibile, spostando i segmenti caricati tutti da una parte), succede, ad un certo punto, che non si riescono più a caricare nuovi segmenti: lo spazio disponibile, in totale, ci sarebbe, ma è diviso in tanti frammenti non contigui, che non possiamo usare, per quanto detto prima.

La paginazione risolve il problema: la memoria lineare non è più la memoria fisica, e possiamo farla apparire contigua, ma caricare le pagine che la compongono come meglio vogliamo. In altre parole, non siamo costretti a caricare tutto il segmento a indirizzi contigui: ogni pagina che compone il segmento la possiamo piazzare ovunque ci sia spazio in memoria fisica (allocando e deallocando la memoria a pagine, saremo anche sicuri che gli spazi disponibili avranno sempre dimensioni multiple di una pagina); poi, mappiamo in memoria lineare le pagine così piazzate, in modo che appaiano di nuovo contigue.

Se ho la paginazione disattivata, non è possibile avere un segmento più grande della dimensione della memoria fisica, vero?

Sì

Pagina 165, volume IV: "*comunemente, tutti i segmenti della memoria logica di un processo sono presenti nella memoria lineare*". Perché? Non esiste il rimpiazzamento di segmenti che permette di ovviare a questa cosa?

Sì, ma *comunemente* vuol dire, appunto, comunemente: anche se la possibilità di rimpiazzare i segmenti esiste, comunemente non viene sfruttata, perché i programmi non ne hanno bisogno (in quanto si riesce ad allocare tutti i segmenti di un programma all'interno dei 4GiB di memoria lineare).

Segmentazione con paginazione attivata: supponiamo che i segmenti di un processo non riescano a stare tutti nella memoria lineare. La frammentazione della memoria lineare è possibile in questo caso?

Certo.

Se devo caricare un segmento, lo devo caricare per intero nella memoria lineare, vero? ...

Esattamente: *caricare un segmento*, in questo contesto, significa trovare in memoria lineare un range di indirizzi R, sufficiente a contenere l'intero segmento, quindi impostare le tabelle delle pagine (o le strutture dati preposte a questo scopo) corrispondenti a tutte le pagine di cui è composto R, in modo che puntino ai corrispondenti blocchi (su disco) che contengono il segmento.

... poi se le sue pagine sono presenti o meno in memoria fisica è un altro discorso, ma comunque in memoria lineare dobbiamo rimpiazzare uno o più segmenti per liberare spazio per il segmento da caricare (per intero), giusto?

Sì

Perché nella *GDT* occorre conservare pure il limite del descrittore? Se ha una struttura rigida non si sa già la sua dimensione? Lo stesso per la *IDT*. Perché conservare in *IDTR* pure il limite? il numero di entrate si sa, la dimensione di ognuna pure.

No, il numero di entrate non è stabilito dall'hardware, solo il numero massimo è fissato.

Potrebbe farmi un esempio di utilizzo dell'istruzione UO presentata nella pagina 144 del volume IV e quindi spiegarmi come funziona?

Forse c'è un equivoco: non esiste nessuna istruzione "UO". Legga così: *Si consideri una istruzione PIPPO che prevede un operando in memoria...*. Ad esempio, prendiamo *PIPPO* = `MOV`. Il paragrafo dice che, per usare un indirizzo logico, il programmatore deve prima caricare un descrittore di segmento in *DS/ES/GS* e poi riferilo nell'istruzione (come visto a reti logiche), ad esempio:

```
movw  $sel, %gs      # carico il registro selettore
movl  %gs:100, %eax  # copio in %eax 4 byte che si trovano
                     # all'offset 100 all'interno del segmento $sel
```

Notate che anche quando non specifichiamo un registro selettore di segmento (come facciamo, ad esempio, alla prova pratica), stiamo, in realtà, usando implicitamente *DS* per tutti operandi in memoria, *SS* per `PUSH` e `POP` e *CS* per il prelievo delle istruzioni. È il caricatore che provvede a inserire valori opportuni in questi registri, prima che il nostro programma venga eseguito.

È corretto dire che la dimensione di un segmento è uguale al limite contenuto nel suo descrittore più uno?

Solo se la granularità è di byte, altrimenti la dimensione è $(\texttt{limite} + 1) \cdot 4096$ byte.

Quando la MMU effettua una traduzione da indirizzo logico a indirizzo lineare usando solo informazioni presenti in un selettore perché non effettua il confronto fra il campo limite e l'offset dell'indirizzo logico come fa invece quando la corrispondenza coinvolge la *GDT* o la *LDT*?

Ho difficoltà a capire la domanda. Credo lei si riferisca al fatto che la MMU (per quanto riguarda la segmentazione) esegue la traduzione usando i valori base e limite presenti nei campi nascosti dei registri selettori di segmento. Vorrei fosse ben chiaro che:

- tali valori vengono necessariamente da una una entrata della *GDT* o della *LDT;*
- la traduzione avviene *sempre* tramite i campi nascosti (quindi il limite viene confrontato, eccome) infatti, per usare un segmento, in questo processore, bisogna necessariamente caricare il selettore S del segmento in un registro selettore, R, (il che comporta il caricamento dei campi nascosti di R con i valori presi dalla *GDT*, o dalla *LDT*, relativi al selettore S), e poi riferirsi al registro R nei successivi accessi al segmento.

Nel caso di un ambiente segmentato l'indirizzo del buffer passato alle primitive di I/O è logico o lineare? L'intestazione della funzione cambia?

In ambiente segmentato, tutti gli indirizzi sono logici. Gli indirizzi lineari non sono visibili al programmatore. Per quanto riguarda l'intestazione della funzione, la risposta è: *dipende*. Il linguaggio C/C++ non prevede costrutti per gli ambienti segmentati, quindi, in tali sistemi, si usano in genere estensioni non standard del linguaggio. Ad esempio, quando nei PC compatibili IBM esisteva soltanto quello che oggi chiamiamo *modo reale* (quello che avete studiato a reti logiche), i compilatori C/C++ della Microsoft richiedevano che, per ogni puntatore, si specificasse, nella dichiarazione, se doveva essere *near* (puntatore all'interno dello stesso segmento) o *far* (puntatore verso un altro segmento). I segmenti venivano poi decisi al momento della compilazione/collegamento.

Immaginiamo di essere in uno spazio segmentato e paginato (farò ipotesi molto semplificative). Ho soltanto due segmenti in memoria logica: il *segmento1* costituito dalla *pagina1* e dalla *pagina2*, e il *segmento2*, costituito dalla *pagina3* e dalla *pagina4*. La memoria lineare è grande due pagine, in questo modo la parte dell'indirizzo lineare che seleziona il descrittore di pagina sarà soltanto il bit più significativo. Se in memoria lineare ho il *segmento1*, un indirizzo di tipo `0` − − − − − − − − − − − − − − − deve individuare il descrittore della *pagina1*, mentre un indirizzo di tipo `1` − − − − − − − − − − − − − − − deve individuare il descrittore della *pagina2*. Quindi il primo descrittore nella tabella delle pagine sarà quello di *pagina1*, e il secondo quello di *pagina2*. Nei descrittori ovviamente avrò l'indirizzo fisico di pagina se la pagina è in memoria fisica, oppure le coordinate per raggiungere la pagina in memoria di massa. Ora, se porto in memoria lineare il *segmento2* (sono costretto a fare swap con il *segmento1*), la tabella delle pagine dovrà avere al primo posto il descrittore di *pagina3* e al secondo il descrittore di *pagina4*? Vorrei sapere se il mio ragionamento è giusto o meno.

Sì, è giusto.

Nel caso in cui sia attiva la paginazione, la memoria lineare a disposizione dei vari processi è sempre la stessa?

È un modello possibile (anche in presenza del campo *CR3* in ogni *TSS: basta* scrivervi lo stesso valore in tutti). Assumiamo che il modello sia questo (unica memoria lineare per tutti i processi).

Ogni processo che ha una sua memoria logica, ha a disposizione tutta la memoria lineare (4GiB), oppure no?

Ogni processo ha a disposizione esclusivamente la sua memoria logica, vale a dire i suoi segmenti. Il sistema, però, ha accesso a tutta la memoria lineare, e può quindi allocare i segmenti di ogni processo ovunque in memoria lineare. Può anche sovrapporre due segmenti, appartenenti a due processi distinti (per esempio, assegnandogli la stessa base e lo stesso limite). Quindi, non è detto che

le pagine che vengono assegnate ad un segmento di un dato processo (al momento in cui questo viene caricato in memoria lineare), non risultino disponibili per segmenti di altri processi.

Non so se ho capito bene come funziona la segmentazione, quando è attivata anche la paginazione. Io credo che sia così: ogni segmento facente parte della memoria logica di un programma è suddiviso in pagine, le quali si trovano in memoria di massa (una copia per ogni pagina). Nel caso in cui la memoria logica sia più grande di quella lineare, non tutti i segmenti possono trovarsi in quest'ultima. Il trasferimento di un segmento in memoria lineare consiste nell'assegnare indirizzi lineari (virtuali) alle pagine che lo costituiscono ed, in pratica, si fa copiando nelle entrate delle tabelle delle pagine, individuate dagli indirizzi lineari appena assegnati alle pagine del segmento, i descrittori ad esse relativi ed aggiornando il descrittore di segmento nella tabella *GDT* o *LDT*, scrivendovi il nuovo valore per il campo base e settando il bit *P*. L'operazione descritta sopra può comportare un rimpiazzamento, dato che gli indirizzi lineari assegnati alle pagine del segmento appena caricato, in precedenza, probabilmente, individuavano pagine di un altro segmento, i cui descrittori vengono sostituiti da quelli relativi alle pagine del nuovo segmento. (quindi, l'entrata di una tabella delle pagine, può trovarsi, in tempi diversi, ad ospitare descrittori relativi a pagine differenti). Funziona così?

Sì, è esattamente così.

Capitolo 14

Multiprogrammazione e Protezione in ambiente segmentato

Un processo utente potrebbe comandare un trasferimento in DMA da una zona di memoria sistema a una utente? Se no, è dovuto al controllo $EPL \geq DPL$ effettuato sui parametri passati alle primitive (in questo caso alla `dmaleggi`)?

Sì, è dovuto al controllo effettuato sui parametri, che, nel caso della segmentazione, coinvolge *EPL* e *DPL* (questo è il problema del cavallo di troia).

Perché i gate di tipo Interrupt/Trap nella paginazione sono visti con un solo bit per il livello di privilegio (*PL*) mentre vedendoli nella segmentazione hanno due bit (*DPL*) per il livello di privilegio? Il tipo del Gate non è uno solo?

Certo che è uno solo. Però, nel capitolo relativo alla paginazione la segmentazione non è stata ancora introdotta, e il professore ha pensato di semplificare, in quel punto, il discorso sui livelli di privilegio (visto che, nella paginazione, ce ne sono solo due).

Perché non possono esservi differenze tra *RPL* e *DPL* per i segmenti codice e pila che invece possono esservi per i segmenti dati?

Per lo stesso motivo per cui si richiede che *CS* e *SS* possano essere caricati (tramite `CALL` o `JMP` per *CS*, e tramite `MOV` per *SS*) solo con selettori di segmento il cui *DPL* è uguale al *CPL*, mentre per i segmenti dati (*DS*, *ES*, *FS*, *GS*) è sufficiente che*CPL* sia maggiore o uguale al *DPL*. Il motivo è che, mentre possiamo tenere sotto controllo i dati, non possiamo tenere sotto controllo il codice. Infatti se lei, programmatore di sistema, scrivesse una primitiva di sistema che, a un certo punto, saltasse con una semplice `CALL` ad una routine di privilegio inferiore (quindi, si suppone, scritta da un altro programmatore, magari un normale utente del sistema) non avrebbe alcuna garanzia di veder ritornare il controllo alla sua primitiva (l'utente, per dolo o per errore, potrebbe

non aver inserito alcuna `RET`, o avere programmato un ciclo infinito, etc.). Stessa cosa (in modo indiretto) se, in una primitiva di sistema, lei usasse una pila di privilegio inferiore (e quindi, si suppone, manipolata da normali utenti): se lei esegue una `RET` mentre sta usando questa pila, salta in un punto deciso dall'utente.

Pagina 162, volume IV: "*il DPL del segmento contenente la routine di interruzione (nuovo CPL) deve essere maggiore o uguale al CPL attuale*" Questo ultimo *CPL* è relativo al processore?

A chi se no? il *CPL* è il Current Privilege Level (livello di privilegio corrente), cioè il livello di privilegio del codice che il processore sta attualmente eseguendo.

Potrebbe farmi un esempio sulla risoluzione del problema del *Cavallo di Troia* nella Segmentazione?

Prendiamo ad esempio la primitiva `read_n`. In un sistema segmentato, l'indirizzo `vetti` del buffer (secondo argomento) sarà composto da selettore e offset. La routine `a_read_n`, nel ricopiare i parametri `interf`, `vetti` e `quanti` dalla pila sorgente alla pila sistema, deve esaminare i due bit B_1 meno significativi del selettore S di `vetti` e confrontarli con i due bit B_2 meno significativi del valore di CS salvato nella pila sorgente. Se il valore di B_2 è inferiore a quello di B_1, `a_read_n` deve sostituire B_2 al posto di B_1 in S, prima di ricopiare S in pila sistema. Quando `c_read_n` tenterà di accedere al buffer puntato da `vetti`, dovrà, per forza di cose, caricare il selettore S in un registro selettore. A quel punto, il processore controllerà che $EPL = min(B_2, CPL) \geq DPL$, generando una eccezione di protezione in caso contrario. In pratica, così facendo, facciamo in modo che il processore verifichi che non solo la primitiva `read_n` (livello di privilegio dato da CPL), ma anche chi ha invocato la primitiva `read_n` (livello di privilegio dato da B_2) ha il diritto di accedere al buffer `vetti` (livello di privilegio dato da DPL).

Parte III

Regole di corrispondenza tra C++ e assembler

Come può la seguente funzione (che è funzione della classe di tipo cl e che crea un oggetto cl di nome cla al suo interno):

```
cl cl::elab1(int arre1[], double arre2[])
{ st1 ss; ss.i = 2; for (int i=0;i<4;i++) ss.vi[i] = arre1[i];
  cl cla(ss); for (int i=0;i<4;i++) { cla.s2.vd[i]+=arre2[i]; }
  return cla;
}
```

accedere con l'istruzione `cla.s2.vd[i]` **al membro PRIVATO** `s2` **dell'oggetto** `cla`**?**

Perché non dovrebbe potervi accedere? l'oggetto `cla` è di classe `cl`, e la funzione `elab1` è, appunto, un metodo della classe `cl`.

A Reti logiche abbiamo imparato come il codice operativo di una istruzione in memoria fosse costituito dal primo byte della medesima, ed inoltre a Calcolatori ci è stato detto che tutte le istruzioni che devono essere trattate dalla FPU devono cominciare con il codice di ESCape 11011, ma le possibili combinazioni di 8 bit che cominciano con 11011 sono solamente 8, mentre le istruzioni che devono essere trattate dalla FPU sono di più in numero, come è possibile?

A Reti Logiche avrete fatto un *esempio* di codifica delle istruzioni. Non è certo una regola matematica che impone che il codice operativo debba essere su un byte. Nell'architettura IA32 (processori Intel x86), il codice operativo può essere di uno o due byte, a seconda dell'istruzione.

A pagina 28, volume III: "1 ***bit per il segno del numero,*** 15 ***bit per l'esponente (o caratteristica), rappresentato in traslazione con fattore di polarizzazione*** $2^{14}-1$***, e*** 64 ***bit per il significando (o mantissa), rappresentato (in forma normalizzata) come*** $1.FFF\ldots F$ ***(***$1 \leq$ *significando* $<$ 2***)*****". Come è possibile rappresentare un insieme continuo di reali se la mantissa è compresa tra** 1 **e** 2**? Come rappresento il numero 9? ...**

In nessun modo, con un un numero finito di bit, è possibile rappresentare un insieme *continuo* di numeri (nel senso matematico di *continuo*). La sua domanda è: come è possibile rappresentare numeri al di fuori dell'intervallo $[1, 2)$? È proprio qui che entra in gioco l'esponente. Prendiamo l'esempio del numero 9, che in binario è

$$1001$$

La prima cosa che dobbiamo fare è normalizzarlo, cioè fare in modo che la virgola si trovi dopo il primo 1. Per far ciò, in questo caso, dobbiamo dividere per 8 (spostare la virgola di 3 posizioni a sinistra):

$$1.001$$

In questo otteniamo la mantissa che, evidentemente, è sempre compresa tra 1 (incluso) e 2 (escluso). Per riottenere il numero di partenza, ci basta rimoltiplicare per 8, cioè per 2^3. Poiché, spostando la virgola, moltiplichiamo o dividiamo sempre per una potenza di 2, ci basta ricordare solo l'esponente (3, in questo

caso). Quindi, il nostro numero di partenza (9) può essere ricostruito se ci ricordiamo la mantissa 1.001 e l'esponente 3 (11 in binario).

... cosa si intende per *rappresentato in traslazione con fattore di polarizzazione* $2^{14} - 1$?

È un modo di rappresentare esponenti sia negativi che positivi. In pratica, al vero esponente viene sommato $2^{14} - 1$ (viene *traslato* di $2^{14} - 1$).

... come si fa, avendo il bit di segno, la mantissa su 64 bit e l'esponente su 15 bit a costruire il numero reale rappresentato?

Se abbiamo una rappresentazione (s, m, e) (dove $s = 0/1$ è il segno, $1 \leq m < 2$ è la mantissa ed $0 \leq e < 2^{15}$ è l'esponente), il numero reale rappresentato sarà:

$$(-1)^s \cdot m \cdot 2^{(e-2^{14}-1)}$$

Per essere precisi, bisogna aggiungere che, tipicamente, poiché la mantissa deve essere sempre 1 virgola qualcosa, l'1 iniziale viene lasciato implicito, per risparmiare un bit, quindi m rappresenta solo i bit dopo la virgola e la formula vera è:

$$(-1)^s \cdot 1.m \cdot 2^{(e-2^{14}-1)}$$

Vado a compilare i files con il comando: `g++27main.cc radici.s -o radici`
ma ottengo un errore di riferimento: `main.cc:(.text+0x9d): undefined reference to 'radici'...`

capita anche a me con il file che mi hai allegato per posta.

... così ho provato a sostituire `_radici`, con `radici`, e tutto funziona correttamente. Il problema è qua; dichiarando nel main la funzione `extern"C"` il compilatore non dovrebbe tradurla in `_radici`?

Per vedere come traduce il compilatore le funzioni potresti provare a vedere l'output del

`flag -E [ Preprocess only; do not compile, assemble or link]`
ridirigendo lo standard output su un file
`gcc ... > test ; less test`

Ho fatto anche una prova levando `extern "C"`, e rinominando la funzione nel file `.s` in `_radici__Fv`, ma ancora problemi di linkaggio (*undefined references to radici(void)*). Qualcuno non è che potrebbe darmi qualche delucidazione a riguardo? Perché anche sui libri di testo, sembra usi la mia stessa procedura. Mi sono accorto facendo un po' di prove con il gcc versione 4.1.2 che la traduzione non è proprio uno a uno come specificata nel libro di testo del corso.

Questo perché la corrispondenza tra assembler e c++ che viene spiegata a lezione si riferisce alla versione di gcc 2.7.2.1 (se mi ricordo bene). Dalla 2.7.2.1 alla 4.1 ce ne sono state un po' di versioni.

L' operazione di `aling expr1, expr2` cosa fa in dettaglio? ...

Bisogna tener presente cosa fa, in generale, l'assemblatore: produce una sequenza di byte (per ogni sezione). Tale sequenza andrà a finire in memoria, a partire da un qualche indirizzo I, nel momento in cui il programma verrà eseguito. L'assemblatore produce i byte della sequenza uno dopo l'altro, in base a quanto legge nel file sorgente. Nel frattempo, mantiene il conto, chiamiamolo C, di quanti byte ha prodotto. Quindi, ogni byte B sarà associato ad un certo valore di C (che è il numero di byte generati prima di B). È chiaro che C sarà anche l'offset in memoria del byte B rispetto a I. Se tutto ciò è chiaro, capire cosa faccia `.align expr1, expr2` è semplice: genera un numero di byte sufficienti a far si che il valore corrente di C diventi un multiplo di $2^{\texttt{expr1}}$, (questo su DOS, in altri casi è diverso). I byte (eventualmente nessuno, se C era già multiplo di $2^{\texttt{expr1}}$) generati saranno tutti uguali a `expr2`. Notate che essere multiplo di 2^a significa, in binario, avere gli a bit meno significativi pari a 0.

... Per le istruzioni può avvenire solo dopo un istruzione di salto perché tra un istruzione e l'altra in fase di traduzione non ci possono essere dei buchi, devono essere sequenziali, vero?

Non in fase di traduzione, ma in fase di esecuzione. Il problema è che, se la `.align` non segue un salto incondizionato, gli eventuali byte di riempimento verrebbero interpretati come istruzioni nel momento in cui il flusso di controllo si trovasse a passare da lì. Il problema può essere risolto anche usando il codice dell'istruzione `nop` come riempimento.

Ho avuto un problema durante la compilazione con il djgpp: in pratica ho provato a compilare con il comando `gcc es1.s prova1.cpp -lgpp` dove i file usati sono due creati a partire dalle soluzioni degli esami precedenti (anche copiando le soluzioni il problema persiste). Penso di non aver impostato nulla di sbagliato (il comando lo riconosce, i nomi sono giusti) solo che mi compare un messaggio di errore che penso dipenda dal fatto che il djgpp o il file .cpp (che mi sono solo limitato a copiare) non riesce a trovare i file di libreria definiti con `#include`. L'errore è il seguente:

```
 In file included from prova1.cpp:2: cc.h:1: iostream.h:
No such file or directory (ENOENT)
```

Questa potrebbe diventare una FAQ. Cerchiamo di capire come la cosa funziona, in modo che ognuno possa provare a risolvere i problemi da solo.

VARIABILI DI AMBIENTE Le variabili di ambiente sono comuni variabili, solo che il loro nome e il loro valore è una stringa di caratteri. Queste variabili vanno pensate come definite in uno spazio *globale*, ma, nel caso di Windows, limitatamente a ciascuna *finestra* DOS (ogni finestra ha le sue variabili e non può leggere o modificare le variabili di un'altra). Ogni processo può anche creare altre variabili di ambiente, oltre a cambiare il valore di quelle esistenti e, ovviamente, leggerne il contenuto. Nella shell (o nei file batch che vengono interpretati dalla shell), una variabile può essere definita (o modificata), con il comando `set`. Per esempio: `set MIAVAR="miovalore"`.

Il contenuto di una variabile può essere letto (in realtà, sostituito al suo nome) racchiudendone il nome tra i caratteri "`%`". Per esempio, il comando `echo %MIAVAR%` stampa il valore corrente della variabile `MIAVAR`.

LA VARIABILE PATH Alcune variabili d'ambiente hanno un significato convenzionale. Tra queste, la più importante è la variabile `PATH`, il cui valore è una sequenza di nomi di cartelle, separate (in MS-DOS) da un carattere "`;`" (punto e virgola). Questa variabile ha un significato speciale per la shell del DOS/Windows 9x/Me (o di Windows NT/2000/XP): quando si scrive un comando al prompt, la shell lo scompone in parole, quindi cerca di eseguire un programma che ha il nome della prima parola (a meno che non riconosca la prima parola come un comando interno alla shell, oppure la prima parola non sia un percorso completo), con in più il suffisso .COM, .EXE, .BAT o altro. Tale programma viene cercato nella cartella corrente e in tutte le cartelle elencate nella variabile `PATH`. Ciò permette di usare un programma anche quando non siamo nella cartella in cui si trova il programma.

DJGPP Il programma `GCC.EXE` si trova (normalmente) nella cartella `C:\DJGPP\BIN`. Se tale cartella è elencata nella variabile `PATH`, possiamo invocare `gcc` (MS-DOS non fa distinzione tra maiuscole e minuscole) in qualunque cartella.

NOTA BENE: questo è il modo corretto di operare, mentre è profondamente sbagliato portarsi nella cartella `C:\DJGPP\BIN` e lavorare lì dentro. "`BIN`" vuol dire "`BINaries`", cioè programmi binari, cioè compilati. è una cartella *di sistema* per DJGPP, che non andrebbe modificata.

`gcc` (nella versione contenuta in DJGPP) ha bisogno anche di un'altra variabile di ambiente, il cui nome è `DJGPP`. Tale variabile deve contenere il percorso completo del file `DJGPP.ENV` (tale file permette a `gcc` di trovare i file delle librerie e altre cose che gli servono). Il file `DJGPP.ENV` si trova, normalmente, nella cartella `C:\DJGPP`, quindi il suo percorso completo è `C:\DJGPP\DJGPP.ENV`, e questo deve essere il valore della variabile `DJGPP`. All'interno della cartella `C:\DJGPP` trovate il file batch `SETENV.BAT`. Tale file contiene le seguenti definizioni:

```
set PATH="%PATH%;C:\DJGPP\BIN"
set DJGPP="C:\DJGPP\DJGPP.ENV"
```

Il primo comando aggiunge la cartella `C:\DJGPP\BIN` a quelle già presenti nella variabile `PATH` (perché i caratteri "`%`" fanno in modo che la seconda stringa `PATH` venga sostituita con il suo valore corrente, prima che `set` possa agire), mentre il secondo crea la variabile `DJGGP` che ci serve.

RISOLUZIONE DEI PROBLEMI Ricordare che le variabili di ambiente sono valide solo all'interno della finestra in cui sono definite. Non ha alcun senso fare *doppio click* su `SETENV.BAT`: verrà aperta una finestra DOS che si chiuderà subito dopo. Bisogna aprire una finestra DOS e lanciare il comando `SETENV`:

```
cd C:\djgpp
setenv
```

oppure, direttamente:

```
C:\djgpp\setenv
```

Ricordare, di nuovo, che le variabili di ambiente sono valide solo all'interno della finestra in cui sono definite. Il comando `SETENV` va eseguito in ogni nuova finestra DOS in cui vogliamo lavorare.

Il file `SETENV.BAT` fornito con DJGGP presuppone che DJGPP venga scompattato nella cartella `C:\DJGPP`. Se non è così, bisogna modificarlo, sostituendo la stringa `C:\DJGGP` con il percorso completo della cartella in cui avete scompattato DJGPP.

Se il problema persiste, contattare il medico.

Quando nella FPU maschero un'eccezione agendo sul registro di controllo e arriva un'eccezione come si comporta la FPU? Continua a lavorare con gli operandi che ha anche se sono errati? Passa alla prossima istruzione? O che?

Produce un valore *ragionevole* e continua a lavorare. Per esempio, nel caso di divisione per 0 di un numero diverso da 0, produce un valore speciale che rappresenta infinito (con il segno opportuno). Nel caso di stack underflow, produce un altro valore speciale, con il signficato di *numero indefinito*.

La domanda che mi appresto a farle non è direttamente legata alla sua materia specifica, ma alla programmazione in generale. Riterrei quindi corretto anche se lei decidesse di non rispondere ritenendola non inerente. Nello studio universitario, dopo aver imparato le classi, nei vari esami, abbiamo usato classi per qualsiasi cosa. Durante il tirocinio sto usando actionscript (stesse specifiche di java). Essendo un linguaggio direttamente legato alla grafica (flash) molte soluzioni e molti programmatori non usano le classi ma si concentrano su funzioni e array. Da poco mi è capitato di dover fare una cosa simile a un mazzo di carte. Ho scelto di usare un array. Poi si è presentato il problema di mischiare le carte. La soluzione adottata è stata definire la funzione shuffle per gli oggetti array. Un'altra possibile implementazione sarebbe stata creare una classe mazzo e una classe carte, e definire la funzione carta, una funzione per inserire una carta e una per estrarre una carta. Pur comprendendo le possibilità aggiuntive delle classi, quello che mi chiedo è: ha senso in casi così semplici utilizzare le classi? A livello di memoria ho un grosso overhead dovuto alle classi o i compilatori moderni compilano le classi così semplici senza overhead? L'esempio portato è uno dei tanti, ho visto usare una classe anche per 2 coordinate su una scacchiera (x e y), utilizzando la classe esattamente come una union.

Molto spesso si hanno atteggiamenti *religiosi* nei riguardi delle tecniche di programmazione, o addirittura sull'uso di un linguaggio rispetto ad un altro. In realtà, un ingegnere dovrebbe conoscere tutte le tecniche a sua disposizione, con i loro vantaggi e svantaggi, e saper scegliere quella più adatta al suo problema.

Molto più spesso, invece, i programmatori scelgono semplicemente la tecnica con cui sono più familiari, e con quella fanno tutto. Inoltre, anche se alcuni linguaggi facilitano l'uso di una certa tecnica rispetto ad un'altra, la tecnica è indipendente dal linguaggio utilizzato: per esempio, si può modellare un problema ad oggetti, e poi scrivere il programma in C, oppure si può scrivere un programma in Java, ma senza usare una modellazione ad oggetti.

Tenendo presenti queste considerazioni generali, veniamo al suo caso specifico. Lei parla di *classi* intendendo, probabilmente, la programmazione orientata agli oggetti. Questa tecnica mostra i suoi vantaggi soprattutto quando il programma dovrà essere soggetto di future modifiche e ampliamenti.

Avere una classe Carta e una classe Mazzo con dei metodi per estrarre e inserire una Carta serve a nascondere, al resto del programma, il modo in cui le Carte sono memorizzate nel Mazzo. Ciò può essere utile se si prevede che l'implementazione possa cambiare in futuro (una tabella hash al posto di un array, per esempio), oppure se prevediamo di voler fare qualche operazione aggiuntiva ogni volta che una carta viene estratta o inserita, etc. Se si prevede che niente di tutto ciò sarò mai necessario, non è necessario ricorrere alle *classi*.

Quanto all'overhead di una tecnica rispetto ad un'altra, bisogna stare attenti ai *miti*. Per prima cosa, un programma non ha un particolare merito nell'essere veloce, per esempio, se non in relazione a quanto percepito da chi deve usarlo. Inoltre, i fattori che incidono pesantemente sull'occupazione di memoria o sulla velocità non sono quasi mai facili da individuare mentre si scrive il programma (entro certi limiti), ma vanno valutati sul programma in esecuzione, con l'aiuto di strumenti appropriati. Anche qui, molto spesso, ci si concentra su quelle che si chiamano *micro ottimizzazioni* (togliere quattro byte di qua o un istruzione di lì), spesso a scapito della struttura del programma. Normalmente, si scopre che queste micro ottimizzazioni portano a minuscole variazioni percentuali sul comportamento del programma finale.

Volevo fare una domanda: visto che non è mai facile correggere un compito quando gli errori sono i propri, sapreste dirmi la sintassi esatta del comando gcc di djgpp che converte un file .cpp in un .s? Lettieri ci parlò di questo comando a lezione (specificando che non era permesso il suo uso durante il compito :), soltanto che non riesco più a trovare i fogli in cui l'ho scritto.

Posso rispondere direttamente io. La sintassi è:

```
gcc -S file.cpp
```

dove, ovviamente, file.cpp è il nome del file da tradurre.

Come viene tradotta da gcc l'istruzione: `pushl` $ − `84(%ebp)`**? ...**

Una piccola nota su questa domanda: gcc, quando parliamo di assembler, ha ben poco da tradurre. Quella che lei ha scritto (a parte il dollaro, che non ci vuole) è una istruzione del processore, che farà certe cose. L'assemblatore si limita a trasformare la stringa `push` nella sequenza di bit che il processore riconosce come push, e poco altro. In nessun caso dovete pensare che l'assemblatore sia qualcosa di complicato come un compilatore, che invece trasforma il codice che

avete scritto, controlla che sintatticamente sia corretto e, in alcuni casi, riesce ad avvisarvi quando scrivete qualcosa che è sospetto dal punto di vista semantico.

È equivalente all'utilizzo delle due istruzioni: `leal` $-$ `84`(%`ebp`), %`eax` **e poi** `pushl` %`eax`**?**

No, `pushl` $-$ `84`(%`ebp`) copia in pila il *contenuto* della parola lunga che si trova in memoria all'indirizzo $-$`84`(%`ebp`). La sequenza di istruzioni da lei scritta, invece, copia in pila *l'indirizzo* di quella parola lunga (vale a dire, il numero che si ottiene sommando -84 al contenuto del registro %`ebp`).

Anche se non fa parte del programma studiato non ho potuto far meno di pensare a come possa essere implementato (a livello di record di attivazione, credo) il meccanismo di protezione dei campi e delle funzioni `private` **e** `protected` **delle classi C++. In che modo sono regolati gli accessi a tali variabili? Vi sono meccanismi aggiuntivi oltre una diversa ed opportuna costruzione del record di attivazione?**

L'accesso ai campi protected e private non è gestito a tempo di esecuzione (e non comporta alcuna modifica al frame di attivazione). Tutte le informazioni necessarie al controllo degli accessi a tali campi sono disponibili in fase di compilazione:

- il compilatore sa che sta traducendo il metodo `X` della classe `Y`;
- inoltre, se il compilatore legge "`A.B`" in una espressione:
  - `A` deve essere stato precedentemente dichiarato nel file che il compilatore sta traducendo (quindi, il compilatore conosce il tipo di `A`);
  - il tipo di `A` (classe o struttura) deve essere stato prececedentemente dichiarato nel file che il compilatore sta traducendo (quindi, il compilatore conosce il tipo di `B` e tutti i suoi attributi `private`/`public`/`protected`, `const`, `volatile`, `virtual`, ...);
- se `Y` deriva da `C`, `C` deve essere stata precedentemente dichiarata nel file che il compilatore sta traducendo, lo stesso per `C` e così via.

Quindi, il compilatore è in grado di sapere, mentre traduce, se il metodo `X` della classe `Y` può accedere al campo `B` della classe `A`. Se sì, traduce normalmente, altrimenti produce un errore e si rifiuta di tradurre il file.

Parte IV

Programmare in Java, volume I

Ho incontrato un problema facendo un esercizio Java. In pratica devo controllare se l'utente batte il ritorno carrello invece di effettuare una scelta dal menù iniziale. Ho provato tutte le possibili letture di stringhe e caratteri (o linea) previste dalla Console ma non ho avuto risultati. Il problema sorge quando, dopo aver fatto la lettura, devo controllare se è stato inserito "\r" o "\n" con un if (e magari se è stato battuto ritorno carrello effettuare anche una semplice scrittura per avere conferma che la condizione dell'if è stata valutata con successo). Finora ho avuto o la generazione di un'eccezione o, anche se il ritorno carrello viene stampato a video non viene considerato e il programma attende che inserisca un carattere buono (a seconda del metodo usato).

Se usa un metodo per leggere una stringa o un carattere, questi salteranno eventuali *spazi* all'inizio della sequenza di caratteri inseriti da tastiera. Per *spazio* si intende il carattere " ", la tabulazione `\t` o anche i caratteri `\n` e `\r`. Se la sequenza inserita è composta solo da questi caratteri, verranno ignorati e l'operazione di lettura si rimetterà in attesa che venga inserito ancora qualcosa. Per leggere una linea qualunque (una qualunque sequenza di caratteri terminata da `\n`), occorre usare il metodo `Console.leggiLinea()`. Questo metodo, però, restituisce la linea inserita da tastiera *con il carattere "`\n`" rimosso*. Quindi, per esempio, il programma:

```
class Prova {
   public static void main(String[] args) {
      Console.scriviStringa("Inserisci una linea");
      String l = Console.leggiLinea();
      Console.scriviIntero(l.length());
   }
}
```

Stamperà 0 (zero) se, al prompt, ci limitiamo a premere invio. Per i suoi scopi, può usare il controllo `l.length() == 0` per sapere se l'utente ha premuto solo invio.

In Java un thread come fa ad acquisire il lock su un oggetto che vuole usare? Cosa significa che acquisisce il lock? Semplicemente che vuole usarlo?

Vuol dire che invoca un metodo synchronized su quell'oggetto.

Informazioni sensitive: perché un'informazione non dovrebbe essere serializzata? Mi può fare un esempio di utilizzo?

L'esempio classico è quello delle password. Un programma potrebbe chiedere una password all'utente e memorizzarla in un campo di un oggetto. Se quell'oggetto viene serializzato, la password verrà copiata in chiaro su un file (che può essere letto anche con altri strumenti, per esempio un comune editor di testo) in pratica svelando la password a chiunque abbia accesso al file. Per evitare questo problema, basta dichiarare `transient` il campo destinato a contenere la password.

Nella correzione mi trovo fermo davanti a questi due errori, se può dirmi da cosa possono dipendere le sarei grato.

```
Pesa.java:55: unreported exception java.lang.Exception;
must be caught or declared to be thrown
          Esa es = new Esa(20);
                   ^
```

il compilatore sta dicendo che il costruttore della classe Esa può lanciare l'eccezione Exception, ma il metodo (nella classe Pesa) che lo usa nè la intercetta nè dichiara di lanciarla a sua volta. Probabilmente lei ha dichiarato che il costruttore di `Esa throws Exception`. Non vedo perché dovrebbe, visto che deve soltanto allocare un vettore, quindi lo levi.

```
Pesa.java:57: unreported exception java.lang.Exception;
must be caught or declared to be thrown
          es.leggi("miofile.dat");
```

Come sopra, lei ha probabilmente dichiarato che il metodo leggi `throws Exception`. Da quello che deve fare il metodo, però, direi che al massimo può lanciare `IOException`, quindi la dichiarazione deve essere aggiustata.

Parte V

Errata Corrige

Il processo Dummy termina quando `lavoro == 1` vero? La `activate_p` viene chiamata anche per attivare il processo dummy? (Nel nucleo scaricabile dal sito, viene fatto così).

Sì, è un errore

A pagina 135 del volume IV si vede che nel file `io.s` sono presenti anche i sottoprogrammi di interfaccia per l'invocazione delle primitive di I/O `readse_n`, `readse_ln`, `writese_n` e `writese_0`. Non dovrebbero questi sottoprogrammi essere invece nel file `utente.s`?

Sì

Perché nello schema della piedinatura del processore non compare il piedino *WT* citato nell'esposizione dei descrittori di pagina in riferimento al campo *PWT*?

Una svista.

A pagina 31 del volume IV viene specificato che il descrittore di processo non contiene i registri speciali, ma a pagina 44 si parla invece della presenza dei registri *CPL* e *CR3*. Quando invece (nel capitolo 10) viene mostrata la struttura di un *TSS* non c'è traccia di *CR3*.

Si tratta di un errore nel libro (dovuto al riarrangiamento degli argomenti rispetto al libro precedente). Nel *TSS CR3* c'è. Punto. A pagina 31 si intende probabilmente parlare dei registri *CR0*, *CR1* e *CR2* (che, effettivamente, non si trovano in *TSS*, anche perché il loro significato è globale), mentre per il capitolo 10 non so. Si tratta comunque di un errore (riconosciuto).

A pagina 83 del volume IV in *sistema.s* non ci andrebbe `.GLOBAL` al posto di `.EXTERN` ?

Sì.

A pag 135 del volume IV nel riepilogo del codice: ...

```
#io8.s
...
.DATA
.GLOBAL _comi
.GLOBAL _como
_comi: .SPACE BYTE_SE
_como: .SPACE BYTE_SE
...
```

... non e errato? Non dovrebbe essere:

```
#io8.s
...
.DATA
.GLOBAL _com
```

```
_com: .SPACE BYTE_SE
...
```

Sì, è errato.

***Nel caso di memoria cache con cardinalità 4 si può dimostrare che il campo R può essere costituito da due soli bit.* La frase mi lascia in crisi, quando si parla delle cache si è detto che l'algoritmo usato per il rimpiazzamento è l'LRU mentre per il TLB è lo pseudo LRU. Io pensavo che venisse usato lo pseudo per diminuire i bit del campo R rispetto al LRU normale. Ma in realtà siamo a 3 bit (pseudo) contro 2 bit (LRU). Allora quale è la vera motivazione per cui si usa lo pseudo LRU per il rimpiazzamento e non LRU?**

È un errore nel libro:

- i bit necessari sono tre, e non due;
- in ogni caso, la frase è fuorviante, perché in realtà vuole dire che possiamo realizzare il campo R con tre bit (come di fatto avviene, ad esempio nel TLB), ma non che riusciamo così a realizzare LRU: con tre bit possiamo realizzare al massimo uno pseudo LRU.

Nel libro sta scritto: la paginazione della tabella delle pagine, che comporta trasferimenti tra memoria di massa e memoria centrale di tabelle delle pagine, comporta che questa risieda a indirizzi virtuali diversi da quelli in cui risiede la memoria fisica, le cui pagine devono essere sempre presenti. Il professore con tabelle delle pagine intende tabella di corrispondenza?

Sì, la frase *la paginazione della tabella delle pagine* dovrebbe essere, in realtà, *la paginazione della tabella di corrispondenza*.

A pagina 165 del volume IV c'è scritto che si usa *CR1 per* copiarci l'indirizzo lineare da tradurre in fisico (paginazione); ma dato che è un page-fault non andrebbe in *CR2* (come descritto nei primi paragrafi, contenente l'indirizzo virtuale non tradotto)?

Sì, è *CR2*.

Nel caso del modello *flat*, perché per avere pagine con livello di privilegio sistema o utente i segmenti dati, codice e pila devono avere privilegio sistema?

Si tratta di un (doppio) errore nel libro, che dovrebbe essere corretto nelle prossime edizioni. Ricordo, intanto, che il modello flat è quello che realizziamo quando non vogliamo usare la segmentazione, anche se il processore (come quello Intel) ci costringe ad usarla. La frase citata nel libro dovrebbe essere corretta: siccome vale il più restrittivo dei due meccanismi di protezione, i segmenti dati e pila devono trovarsi a livello *utente* (in modo che le restrizioni della segmentazione non abbiano effetto e valgano solo quelle della paginazione).

C'è poi l'altro errore, che riguarda la realizzazione del modello flat: se è attiva la segmentazione, in questo processore, il livello di privilegio corrente è comunque deciso dal contenuto del registro *CS*. Quindi, se vogliamo avere i due livelli di privilegio (utente e sistema) che ci servono per impedire l'accesso a certe pagine tramite il bit *S/U*, dobbiamo avere *due* segmenti codice, sempre sovrapposti e grandi 4GiB: uno con *DPL* utente e l'altro con *DPL* sistema. In *CS* ci sarà l'uno o l'altro, a seconda del livello di privilegio corrente del processore. Di conseguenza dovremo avere anche *due* segmenti pila (uno a livello utente e uno a livello sistema). Non è necessario, invece, avere alcun segmento dati (si possono usare gli stessi usati per la pila).

A pagina 51 del volume IV c'è scritto che il `des_proc` dovrebbe contenere tra le altre cose un campo precedenza, ma questo non compare da nessuna parte (nemmeno in altre parti dove è riscritto il `des_proc`). Come mai?

È un errore. Il campo precedenza è stato spostato da `des_proc` a `proc_elem` (dove ora si chiama `priority`). Concettualmente non cambia niente: si tratta, in entrambi i casi, di associare una priorità ad ogni processo.