1 Gestione della memoria

Abbiamo già detto che la memoria virtuale è un meccanismo, in parte gestito dall'hardware ed in parte dal software, che permette ai programmatori di avere a disposizione uno spazio di indirizzi di memoria molto più vasto della memoria fisicamente disponibile su computer su cui il programma gira.
Il compiti del modulo di gestione della memoria del S.O. ("**memory manager**") sono quindi relativi a due classi di servizi:
- realizzazione e gestione ottimale della memoria virtuale
- gestione dell'assegnazione della memoria virtuale ai processi

In questo capitolo espliciteremo alcuni dettagli sulle tecniche di realizzazione della memoria virtuale e sul relativo supporto hardware delle CPU X86.

1.1 Memoria virtuale

Per svolgere la sua funzione la memoria virtuale deve possedere un meccanismo che "traduce" in qualche modo indirizzi virtuali in indirizzi fisici di memoria RAM (Figura 1).

<FILE>
trasformaz.FH5
<FILE>

Figura 1: trasformazione di indirizzi

Il meccanismo della memoria virtuale si appoggia a tabelle di conversione che, almeno a livello di principio, possono essere viste come le tabelle in Figura 2.

<FILE>
tabelle.FH5
<FILE>

Figura 2: tabelle di conversione

Come vedremo, in realtà le tabelle hanno una struttura parecchio più complicata. Inoltre il meccanismo della memoria virtuale, per avere un'efficienza accettabile, deve essere supportato dall'hardware della CPU e le tabelle, per essere di dimensioni praticabili, devono gestire "blocchi" continui di associazioni fra indirizzi virtuali e fisici.
Si tratta comunque di un meccanismo trasparente all'utente gestito fra hardware e software in modo completamente automatico.
Il programmatore fa uso di indirizzi virtuali, che vengono tradotti automaticamente da parte di CPU e S.O. nei corrispondenti indirizzi fisici. Egli non ha visione della memoria virtuale e potrebbe benissimo ignorarne l'esistenza, dato che tratta la memoria virtuale come se fosse "molta" memoria fisica.
Per esempio, un programma con 386 può eseguire un'istruzione come la seguente:

```
MOV AL, [LocMemVirtuale]
```

Ove LocMemVirtuale è un indirizzo di memoria virtuale. La CPU, consultando le tabelle della memoria virtuale, è in grado di trovare il corrispondente indirizzo fisico e di leggere da esso il byte da trasferire in AL.
Dato che lo spazio di indirizzamento virtuale è molto più ampio di quello fisico i programmi potranno richiedere ed ottenere più memoria di quanta ne sia effettivamente presente sul computer.
Perché questo sia possibile bisogna che una parte della memoria virtuale, in genere quella non utilizzata dal processo corrente, venga letta e copiata sull'hard disk, e lì lasciata fino a che non servirà nuovamente.
Dunque le tabelle che realizzano la corrispondenza fra memoria fisica e virtuale possono contenere l'indicazione che l'indirizzo virtuale cercato non è attualmente ospitato nella RAM. (vedi Figura 2, riga XXXX).
Se c'è la necessità di usare, in lettura od in scrittura, il contenuto di una locazione di memoria virtuale attualmente sull'hard disk, il S.O. deve effetture lo swap-in.

Vediamo cosa accade in questo frangente in una CPU X86 (286>).
Supponiamo che il nostro programma debba leggere il valore corrente di una certa locazione virtuale (LocMemVirtuale).

```
MOV EAX, [LocMemVirtuale]     ; esempio per CPU 386 >
```

Supponiamo anche che le tabelle per l memoria virtuale, che studieremo in dettaglio nel seguito, indichino alla CPU che l'indirizzo LocMemVirtuale non ha attualmente un corrispondente in memoria fisica.
La CPU si accorge, consultando le tabelle, che la locazione virtuale richiesta è attualmente nello swap file e lancia un'eccezione, che chiameremo "page fault" (la ragione della denominazione sarà chiara più avanti).

L'eccezione corrisponde ad un preciso vettore del sistema d'interruzione della CPU. Alla locazione cui corrisponde quel vettore si trova una procedura che fa parte del S.O., che nella Figura 3 è stata indicata con l'etichetta "VirtMemManager".

<FILE>
PageFault.FH5
</FILE>

Figura 3: eccezione di mancanza di una pagina

L'indirizzo di VirtMemManager sta nella tabella dei vettori d'interruzione, ed è accessibile tramite l'INT# dell'eccezione, che è fissato dal produttore della CPU (es. L'INT # dell'eccezione di page fault nel 386 è XXXX).
La ISR dell'eccezione di page fault, che è parte del gestore della memoria del S.O., deve:
trovare posto in memoria fisica per il blocco di memoria virtuale che deve "entrare",
decidere, se è il caso, quale blocco debba lasciare la memoria fisica per essere parcheggiato sull'hard disc
effettuare le operazioni di swap out e swap in dei blocchi interessati
aggiornare le tabelle della memoria virtuale, che debbono tener traccia della nuova situazione.
Quando lo swap in del nuovo blocco è terminato la locazione di memoria virtuale LocMemVirtuale è effettivamente in memoria fisica, per cui l'istruzione IRET può tornare alla stessa istruzione che ha causato l'eccezione.
L'istruzione viene di nuovo eseguita e questa volta LocMemVirtuale è in memoria fisica e da lì può essere letto ed eventualmente modificato, non l'eccezione non scatta più e le tabelle della memoria virtuale restituiscono l'indirizzo fisico su cui la MOV potrà effettivamente agire.

1.2 Segmentazione e paginazione

Le due tecniche più comuni per la realizzazione della memoria virtuale sono la segmentazione e la paginazione.
Le due tecniche hanno caratteristiche abbastanza diverse, che le rendono adatte per applicazioni diverse.
Entrambe suddividono la memoria virtuale in "blocchi" di locazioni contigue, che all'occorrenza possono essere trasferiti in blocco :- | fra memoria fisica ed hard disc.

La **segmentazione** (segmentation) divide la memoria virtuale in blocchi di lunghezza variabile.
La **paginazione** (pagination), divide la memoria virtuale in blocchi di lunghezza fissa.

Il fatto che una tecnica preveda blocchi variabili e l'altra blocchi fissi ha pesanti implicazioni sulla struttura delle tabelle della memoria virtuale e sul loro contenuto, come vedremo nel paragrafo che riguarda le CPU X86.
Segmentazione e paginazione si possono usare insieme (segmentazione con paginazione); le CPU X86 dal 386 in poi mettono a disposizione dei meccanismi hardware che semplificano il lavoro del progettista del S.O. nella gestione dei sistemi, segmentati, paginati od anche con entrambe le modalità di funzionamento.
Trascurando per il momento come avviene la trasformazione fra indirizzo virtuale e fisico, analizziamo le caratteristiche delle due tecniche riguardo al fenomeno della frammentazione.

1.2.1 Frammentazione

Si definisce "frammentazione" (fragmentation) il fenomeno per il quale, durante l'uso della memoria virtuale, si creano delle zone della memoria fisica che non sono utilizzabili.
La frammentazione crea un problema di efficienza del sistema, perché la memoria viene "sprecata".

Frammentazione esterna
La Figura 4 illustra il fenomeno della frammentazione nel caso della memoria virtuale segmentata. I segmenti, essendo di dimensione variabile, possono essere allocati della stessa dimensione che serve.
La situazione illustrata inizialmente mostra la memoria fisica con i segmenti allocati in zone contigue della memoria; la memoria è quasi completamente utilizzata.
Questa è la situazione che si potrebbe presentare all'inizio del funzionamento del computer, quando la memoria RAM non è stata ancora usata fino alla sua massima capacità.

<FILE>
FrammEsterna.FH5
</FILE>

Figura 4: segmentazione e frammentazione

In Figura 4 il punto (1) indica la richiesta dell'allocazione del segmento S5 da parte di un processo.
Quel segmento non può essere contenuto nella piccola area di RAM rimasta libera, per cui il gestore della memoria deve decidere lo swap out dei due segmenti S2 ed S3 (2).
Poi parte dell'area liberata viene destinata a contenere S5 (punto (3) di Figura 4, parti (a) e (b)). Siccome S5 è più piccolo della somma fra S2 ed S3 una parte della memoria, alla fine di S5, rimane libera.

Questa memoria è frammentata, perché essendo di piccole dimensioni, difficilmente potrà contenere un intero segmento e rimarrà inutilizzata.
Immaginiamo infatti che il segmento S3 debba rientrare in memoria fisica, come evidenziato dal punto (4) di Figura 4, nella parte (b). Nella situazione illustrata nella parte (b) la memoria libera è maggiore della lunghezza di S3, ma si è costretti all'operazione di swap out di S4 perché quella memoria libera è frammentata (punto (5) di Figura 4).
Una volta che il segmento S4 ha liberato la memoria, S3 può subire uno swap in e rientrare nella RAM (punto (6) di Figura 4, parti (b) e (c)).
Si capisce come, avendo centinaia di segmenti ed effettuando frequenti operazioni di swap, la memoria diventi tutta "bucherellata" e diminuisca il suo grado di utilizzazione.
Naturalmente è possibile applicare delle tecniche di compattazione della memoria, che eliminino la frammentazione quando essa diventa troppo pesante. Questo peraltro richiede tempo CPU da dedicare alla compattazione, diminuendo l'efficienza del computer.
Il tipo di fenomeno appena illustrato viene detto "**frammentazione esterna**", perché le aree di memoria che divengono inutilizzabili sono esterne ai blocchi che devono essere allocati.

Frammentazione interna
Supponiamo ora di disporre di un sistema di memoria virtuale paginata, nel quale quindi i blocchi sono tutti della medesima dimensione.
Quando un processo richiede della memoria, potrà ottenerne solo un certo numero di pagine, quindi si può dare il caso che ottenga un po' più di memoria di quanto richiesto; se per esempio le pagine sono di 4 kByte ed un processo richiede 10 kByte, otterrà 3 pagine, per un totale di 12 kByte; due kByte saranno inutilizzati.
Questo delinea un problema detto di "**frammentazione interna**", perché la memoria non utilizzata è all'interno dei blocchi di memoria effettivamente allocati al processo.
Con molti processi che richiedono molte aree di memoria la frammentazione interna può divenire un problema considerevole, anche se è chiaro che la massima frammentazione per ogni area di memoria allocata corrisponde alla dimensione della pagine meno un byte, cioè al caso più sfortunato in cui di deve allocare una pagina in più perché la memoria che ci serve è di un solo byte in più di un multiplo della dimensione della pagina.
Dunque per rendere trascurabile il problema della frammentazione interna basta usare una dimensione della pagina ridotta. Per contro se la dimensione della pagina è troppo piccola la dimensione delle tabelle per la memoria virtuale diviene troppo grande e la loro gestione troppo onerosa, rendendo il sistema inefficiente.

<FILE>
FrammInt.FH5
</FILE>

Figura 5: swap di memoria virtuale paginata

La Figura 5 mostra una memoria virtuale paginata "piena" e con frammentazione interna. Nel punto (a) avviene la richiesta di tre pagine di memoria da parte del processo P4. E' chiaro che è necessario lo swap out di 3 pagine. Supponiamo che il memory manager decida che debbano uscire Pr1pag2, Pr2pag1 e Pr3pag1 (punto (2); questa è una decisione improbabile, ma non impossibile!).
La situazione in Figura 5, parte (b), mostra che il processo P4 ha tutta la memoria virtuale richiesta attualmente in RAM, mentre P1 e P3 hanno parte della loro memoria virtuale presente in RAM e parte parcheggiata nello swap file.
Supponiamo ora che il process scheduler mandi in esecuzione il processo P1, il quale richieda il contenuto di una locazione di P1pag2, attualmente sull'hard disk. Questa è una situazione nella quale si genera un "page fault" con il quale il processo richiede la pagina mancante al S.O. ("**demand page**").
In conseguenza del page fault il S.O. inizia un processo di swap, che porta fuori P3pag2 (punto (4) di Figura 5) e dentro P1pag2 (punto (5) di Figura 5) ("page replacement").
È interessante notare che ora tutta la memoria virtuale richiesta da P1 si trova in RAM, ma P1pag2 ha cambiato indirizzo fisico rispetto alla condizione iniziale, naturalmente senza cambiare indirizzo virtuale!
Questa è una situazione del tutto normale con la paginazione, nella quale gli indirizzi di memoria virtuale visti dai processi appartengono ad un blocco sequenziale, mentre gli indirizzi fisici corrispondenti sono sparsi o addirittura inesistenti.
Dunque non tutte le pagine di un processo in esecuzione devono essere necessariamente in memoria fisica; al limite potrebbero essere caricate anche solo quella ove è il codice che sta eseguendo e quella con i dati che si stanno modificando, che potrebbero anche coincidere.
Analizzando la Figura 5 ci si può anche rendere conto di come, con pagine di dimensione fissa "swappabili" singolarmente, il problema della frammentazione esterna non esista.

In alcuni casi il S.O. o la CPU tengono traccia, con un bit nelle tabelle delle pagine, di quando il contenuto dalla pagina viene modificato (bit "**dirty**" (sporco)). Se questo bit indica che la pagina non è stata modificata al momento del suo swap out si potrà evitare di copiarla nell'area di swap.
Per limitare il numero di operazioni di swap il S.O. può tener traccia di quali pagine sono state usate in un certo periodo di tempo. Le pagine non usate da molto tempo sono buone candidate per lo swap out.

Per sapere se una pagina è stata usata si può usare un bit nella tabella (bit "**accessed**"). Questo bit viene messo a 1 quando la pagina viene usata, sia in lettura che in scrittura.
Al momento dello swap out il S.O. decide di togliere dalla memoria una delle pagine che hanno di bit accessed a zero.
In questo modo tendono a rimanere in memoria le pagine usate più di frequente e quindi, per il principio di località (vedi Volume1), quelle che hanno maggiore probabilità essere usate ancora.

!!!! trovare posto:
Per limitare il fenomeno del t i S.O. reali pongono un limite alla quantità di memoria virtuale che si può allocare. Questo limite dipenderà dalla quantità di memoria fisica presente e non supererà di molto il doppio della RAM.
I Sistemi Operativi possono decidere di eliminare dalla memoria i job batch se verificano una condizione di t.

Politiche di swapping
!!!! mettere qualcosa o dire di cercare in altri libri ????
Buddy algorithm (algoritmo "del migliore amico")

Gestione della memoria
MEMORY ALLOCATION SCHEMES
Single process system
Operating system
User process
Unused space
Relocation register needed to indicate base address of user process.
For protection, also need register to indicate limit address.
Fixed partition system
Several partitions each capable of holding one program. Partitions might or might not be of same size.
Largest partition needs to be large enough to hold largest program.
Results in internal fragmentation of memory due to unused space () in each partition.
Segmented systems
Program split into two or more (variable sized) segments.
Most common division is code and data. Also possible to have programmer-defined segments (e.g. one for each module of source code).
Segments need not be contiguous, thereby reducing problems of allocating free space.
Need separate pairs of registers for each segment.
Segment attributes: sharable; writeable.
Problems
Fragmentation. Cannot use space between segments unless there is another segment small enough to fit.
Compaction is almost as inefficient as swapping (but at least frees up some useful space).
Trade-off between more, smaller segments making memory fitting easier against need for more segment registers.
PAGING
Memory divided up into fixed-size chunks called .
Typical page sizes: 512 bytes, 1K, 2K, 4K, …
Relocation problem - effectively need a relocation register for each page. However since all pages are the same size, no limit register is needed.
Page addressing
Each process has a cal pages (and any page attributes).
Each logical address is split into two parts: page number and offset within page. Page number mapped by page table, then offset added in.
Solves most of fragmentation problem since a process can be fitted in to a number of separate holes. Some fragmentation still exists because of unused space within a page.

!!!! trovare posto:
Le esigenze delle aree di swap di un disco, contrapposte a quelle che contengono file normali, sono parecchio diverse, tant'è vero che Linux ha un file system specifico solo per lo swap, Windows invece ha un file swap visibile anche dal normale file sstem.

!!!! trovare posto:
Alcuni sistemi in tempo reale non implementano la memoria virtuale, dato che l'esigenza non è particolarmente sentita nei sistemi embedded, ed essa contribuirebbe a rendere più complicato e meno deterministico il comportamento del sistema.

!!!! trovare posto:

I S.O. moderni permettono di "bloccare" in memoria la pagine dei processi più importanti, come per esempio tutti i processi che fano parte del kernel, evitando che subiscano uno swap out.

<AVANZATO>

Rilocazione
RELOCATION
Program with user address space starting at 0 will not necessarily be loaded into a physcal space starting at address 0.
Relocation of addresses (but not data).
Load time relocation (sorted out once when program is loaded - prgram amended).
Runtime relocation via relocation registers (sorted out each time a logical address is acessed).

Supporto dell'hardware
Le odierne CPU aiutano il S.O. nella realizzazione della memoria vituale.
Prendiamo per esempio le CPU della famiglia X86 dal 386 in poi. Come già abbiamo visto le tabella delle pagine per la memoria virtuale paginata iniziano tutte dalla "page directory", che è puntata dal registro CR3.
Il "memory context switch" in un 386 consiste perciò solamente nel cambiare il valore di CR3 con quello del nuovo processo; tutto il resto è gestito direttamente dall'hardware della CPU.
Quindi in un 386 il cambio di contesto di memoria è piuttosto indolore. Nelle CPU più moderne però il discorso cambia, perché sono presenti cache per dati e istruzioni che, nel caso di cambio di processo, non sono più utili e debbono essere ricaricate. Per questo nelle CPU moderne il "memory context switch" è un'operzione piuttosto "costosa".
</AVANZATO>

1.3 CPU della famiglia X86 in modo protetto

Come già detto nella Parte 2, cui si rimanda per le generalità, le CPU X86 dal 286 in poi ammettono un modo di funzionamento, detto modalità "protetta", che fornisce al Sistema Operativo una serie di servizi utili per la sicurezza e per la realizzazione della memoria virtuale.
Le caratteristiche del modo protetto 286 erano piuttosto incomplete, tant'è vero che non furono mai utilizzate dal S.O. dominante in quel periodo (MS-DOS), che lavorava in modo reale e non fu mai adattato per funzionare in modo protetto, a parte qualche "trucco" per poter indirizzare più memoria.
Al contrario il modo protetto del 386 è completo ed efficace ed è, in buona sostanza, quello usato tuttora in tutti i sistemi operativi per piattaforme X86.
In questa parte del nostro testo descriveremo le modalità di funzionamento di un 386 in modo protetto, indicando "a latere" ciò che mancava nel 286.
Nella letteratura che fa riferimento al modo protetto il costruttore delle CPU X86 parla di "task", il termine, che ricorrerà anche nel resto di questo capitolo e che si può considerare equivalente al termine "processo".

1.3.1 Protezione della memoria e delle istruzioni

Privilege levels can be used to improve the reliability of operating systems. By giving the
tected
from damage by bugs in other programs. If a program crashes, the operating system has a
chance to generate a diagnostic message and attempt recovery procedures.

Nella famiglia X86 la protezione riguarda diversi dispositivi e funzioni:
- protezione dagli accessi al di fuori delle zone di memoria consentite
- protezione dall'esecuzione di codice che ha privilegio non sufficiente
- protezione dal danneggiamento del codice
- protezione dall'esecuzione di alcune istruzioni che potrebbero fare danni (p. es. istruzioni riguardanti il sistema delle interruzioni e quelle che usano i registri interni di sistema).
- protezione dalle operazioni sui port di Input Output (istruzioni IN e OUT).

I meccanismi di protezione X86 si basano in parte su come viene organizzata la memoria virtuale, che può essere "segmentata", "paginata" o "segmentata e paginata" (vedi oltre).
Le protezioni che riguardano la segmentazione si basano su un meccanismo a 4 livelli di privilegio, detti "protection **ring**" ("anelli" di protezione).
I quattro livelli sono codificati con un numero binario di due bit, da 0 a 3. Il livello più privilegiato corrisponde al numero 0. Quando la CPU sta eseguendo a questo livello può accedere ad ogni locazione di memoria ed eseguire ogni istruzione. Per contro, se la CPU sta eseguendo a ring 4 avrà i maggiori vincoli di protezione.
Il meccanismo dei privilegi realizza la necessaria separazione fra software di sistema ed applicazioni, il software di sistema funzionerà a livelli di protezione più "bassi" di quelli del software applicativo.
I livelli di privilegio per la segmentazione X86 sono:
Livello 0 (Ring 0):"kernal": il livello di privilegio al quale funziona solo il "cuore" del sistema operativo, che controlla tutto il computer, cioè il kernel. Il kernel è un programma che deve essere efficientissimo e di piccole dimensioni e si interfaccia direttamente con l'hardware del computer e con i sistemi di I/O. Per questo, i programmi di ring 0 non

hanno alcuna limitazione nell'accesso alla memoria: tutto il codice ed i dati presente in memoria può essere raggiunto e modificato quando la CPU esegue un programma in Ring 0. Viceversa, ogni parte di codice o dati che è di ring 0 non è accessibile altro che da programmi che siano di livello di privilegio 0.

Livello 1 (Ring 1): "System Services": è il privilegio che dovrebbe essere usato dai driver dei dispositivi. Se un driver va in crash la parte di S.O. che gira a privilegio 0 può mantenere in controllo del sistema, mentre se un'applicazione di privilegio 2 o 3 va in crash non coinvolge anche i device driver, che continuano a funzionare.

Livello 2 (Ring 2): "Custom Extensions": in questo livello risiedono i programmi "normali" del Sistema Operativo, quelli hanno un contatto meno immediato con le periferiche e che utilizzano un primo livello di astrazione. Il software a questo livello è isolato dalle applicazioni d'utente, ma anche dai programmi che funzionano ai livelli inferiori e per questo non può danneggiarli.

Livello 3 (Ring 3): "Applications": il livello a cui appartengono tutti i normali programmi di utente.

L'accesso ai dati è possibile solo a task che abbiano un livello di privilegio uguale o migliore (numero più basso) a quello del segmento richiesto.

Se un task tenta di accedere a dati per i quali non ha privilegio sufficiente la CPU lancia un'eccezione, in questo modo fa intervenire il S.O, esso è così in grado di isolare il programma che ha generato l'errore e, in genere, di terminarlo.

La protezione riguarda anche alcune istruzioni, che possono essere eseguite solo a certi livelli di privilegio. Sono protette, per esempio, le istruzioni: CLI e STI, IN e OUT.

1.3.2 Supporto alla memoria virtuale nell'architettura X86

Come accennato, in una CPU X86 dal 386 in poi ci possono essere due tipi di memoria virtuale, la memoria virtuale "segmentata" e quella "paginata".

Il programmatore di sistema può decidere:

se non vuole usare la memoria virtuale; avrà a disposizione indirizzi fisici di 32 bit

se vuole usare la memoria virtuale segmentata, avrà a disposizione indirizzi virtuali di 46 bit, corrispondenti a 64 TByte! (30 nel 286, corrispondenti a 2 GByte) (modo protetto)

se vuole usare la memoria segmentata e paginata (segmentazione con paginazione); avrà a disposizione due indirizzi virtuali, uno di 46 bit, l'altro di 32 bit, gestiti automaticamente dalla CPU (modo protetto 386>, nel 286 la paginazione non esisteva).

Il meccanismo della segmentazione funziona sempre, non si può disabilitare a livello di hardware ma si può "eliminare" di fatto in software, utilizzando un modello "piatto" della memoria (vedi, in seguito "flat mode").

Memoria virtuale segmentata

Quando un X86 lavora in modo protetto il meccanismo della segmentazione funziona in modo molto diverso rispetto al caso della modalità reale. Per il momento trascuriamo completamente la paginazione, supponiamo che non esista, o che sia disabilitata; la potremo "aggiungere" in seguito.

I registri di segmento degli X86 sono a 16 bit, anche nei Pentium, ed hanno gli stessi nomi e funzioni analoghe a quelli dell'8086 (DS, CS, SS, ES; FS e GS dal 386).

Solo 14 dei 16 bit dei registri di segmento sono utilizzati per formare l'indirizzo virtuale, gli altri due indicano il livello di privilegio a cui avviene l'accesso protetto a quel segmento (RPL: **R**equestor's **P**rivilege **L**evel), per i livelli di privilegio vedi in seguito.

Quattordici bit del registro di segmento sono la parte alta dell'indirizzo virtuale, li chiameremo "**selettore di segmento**"[1].

<FILE>
RegSegX86.FH5
</FILE>

Figura 6: contenuto dei registri di segmento X86 in modo protetto

La Figura 6 illustra anche una parte nascosta del registro di segmento, che viene gestita in modo del tutto automatico dalla CPU, il cui significato verrà spiegato più avanti.

L'indirizzo che usano tutti i programmi in modo protetto è un indirizzo virtuale di 46 bit (30 nel 286), che non viene emesso sull'address bus e che viene anche chiamato "indirizzo logico", in contrapposizione all'"indirizzo fisico", che è quello effettivamente usato sull'address bus.

L'indirizzo logico è ottenuto come composizione di una parte di segmento, da 14 bit, e di un offset da 32 bit (16 nel 286), come illustrato dalla Figura 7.

Si noti che la parte di segmento e quella di offset non si sovrappongono, come accadeva in modalità reale, e perciò non ci sono le ambiguità di indirizzi già discusse nel Volume 1.

<FILE>

1 in verità il produttore chiama "selettore di segmento" tutti i 16 bit contenuti nel registro di segmento; all'Autore questa definizione non piace perché i due bit di RPL in realtà non "selezionano" nulla.

IndVirt286.FH5 e IndVirt286.FH5
</FILE>
Figura 7: l'indirizzo virtuale X86
Tutte le volte che la CPU deve accedere ad una locazione di memoria virtuale, deve "tradurre" l'indirizzo virtuale di 46 bit in un indirizzo fisico da 32 bit (l'indirizzo fisico del 286 è di 24 bit).
Per la traduzione la CPU si appoggia ad una tabella, situata in memoria fisica, che contiene l'indirizzo fisico della prima locazione di ogni segmento.
Per l'accesso a questa tabella la CPU usa le informazioni che trova nel registro di segmento, mentre lo spostamento fra l'inizio del segmento e la locazione interessata è dato dall'offset dell'indirizzo virtuale.

Il selettore di segmento è un numero usato come indice per puntare ad una locazione fisica in una tabella di sistema, detta "**tabella dei descrittori di segmento**".
Accedendo alle informazioni nella tabella la CPU potrà calcolare l'indirizzo della vera locazione fisica, che utilizzerà per l'accesso all'indirizzo virtuale specificato.
Esistono due tipi di tabelle dei descrittori: globali e locali. Esse si chiamano: **G**lobal **D**escriptor **T**able (GDT) e **L**ocal **D**escriptor **T**able (LDT). La GDT contiene la descrizione dei segmenti di memoria virtuale che possono essere visti da tutti i task, la LDT è relativa ai segmenti visibili al solo task che sta eseguendo sulla CPU in un certo istante.
Lo spazio virtuale d'indirizzamento di un X86 è perciò diviso in due parti, una viene detta "spazio d'indirizzi globale", l'altro "spazio d'indirizzi locale". Tutti i task che eseguono in un S.O. che usi la segmentazione X86 in modo protetto condividono lo stesso spazio d'indirizzi globale, mentre ogni task può avere il ha il suo proprio spazio d'indirizzi locale.
Per distinguere fra spazio globale e locale la CPU usa il bit TI (**T**able **I**ndicator) del registro di segmento. Se esso è a zero, il segmento è nello spazio di indirizzi globale, altrimenti è locale.

I rimanenti bit del selettore di segmento (13 bit) sono detti INDEX e servono per puntare in memoria fisica al giusto elemento, si potrebbe dire al giusto "record", nella tabella dei "descrittori di segmento".
In sostanza INDEX è il numero del segmento dentro la tabella dei descrittori e può andare da 0 a 8192 (13 bit).
Ogni segmento usato in ogni programma deve contenere la sua descrizione in una di queste tabelle. Ogni programma (task) ha la sua tabella di descrittori locale, ove sono definiti tutti i segmenti di memoria virtuale che utilizza. Inoltre il task usa, insieme a tutti gli altri programmi, la tabella, unica, dei descrittori globali.

<FILE>
TabSegmenti.FH5
</FILE>
Figura 8: puntatori alle tabelle dei descrittori di segmento.

L'indirizzo in memoria fisica (di 32 bit (24 per il 286)) dell'inizio delle due tabelle dei descrittori è conservato in due registri del 286, GDTR e LDTR (**G**lobal e **L**ocal **D**escriptor **T**able **R**egister), ed è indicato come "INIZIO" in Figura 8.
Questi registri, oltre che "INIZIO", contengono anche la lunghezza della tabella (20 bit (16)).
Ogni volta che il Sistema Operativo fa un cambio di task deve cambiare anche il contenuto di LDTR, ma non quello di GDTR, che deve continuare a puntare alla memoria "comune" fra tutti i task.
Per sapere che numero mettere in LDTR, diverso in base al task che esegue, la CPU utilizza la GDT, che viene opportunamente gestita dal S.O.[2]

La GDT e la LDT sono costituite da tanti descrittori quanti sono i segmenti utilizzati.

Il descrittore di segmento

Il descrittore ha la lunghezza di 8 byte (4 word) e contiene tutte le informazioni necessarie a definire il segmento. Si possono individuare tre "tipi" di informazione:
la lunghezza del segmento (LIMIT, 20 bit (16))
l'indirizzo iniziale in memoria fisica (BASE, 32 bit (24))
alcune informazioni relative all'accesso al segmento
Nel seguito descriviamo il contenuto di ciascuno dei campi del descrittore di segmento. I descrittori sono leggermente diversi se il programma è un applicativo od un programma di sistema. La Figura 9 mostra i campi contenuti nel descrittore di segmento.

<FILE>
DescSeg386.FH5
</FILE>
Figura 9: Campi contenuti in ogni descrittore di segmento.

2 In realtà i S.O. attuali, che utilizzano il modello "piatto", non usano una LDT per ogni processo, ma solo la GDT, per tutti i processi; la protezione non è ottenuta con la segmentazione ma con la paginazione (vedi oltre).

La lunghezza (**LIMIT**) è un numero di 20 bit (16) che indica l'offset massimo ammissibile per accessi protetti alla memoria che non diano errori (è cioè la dimensione del segmento). Anche se il campo è di soli 20 bit la lunghezza massima dei segmenti non è 1 MByte, come ci si potrebbe attendere, ma 4 GByte. Per avere segmenti così lunghi viene usato il "trucco" della "granularità", che vedremo in seguito. Nel 286, con un campo LIMIT di 16 bit, si raggiunge la lunghezza massima del segmento, di 64 kByte, senza bisogno di trucchi.
Se un task tenta di avere accesso a locazioni di memoria fisica che vanno oltre l'indirizzo d'inizio del segmento più la sua lunghezza, la CPU genera un'eccezione di "errore generale di protezione" ("general protection fault" INT 13).

L'indirizzo iniziale (**BASE**) è l'indirizzo di memoria fisica dal quale parte il segmento. Siccome è di 24 bit, prende 3 byte del descrittore.
Come si può vedere, i campi LIMIT e BASE sono dispersi in diverse posizioni nel descrittore, in parte negli ultimi due byte. La ragione di questa dispersione risiede nel fatto che il descrittore di segmento del 386 è stato mantenuto compatibile con quello del 286, come si può apprezzare dalla Figura 9. In questo modo i programmi scritti per il 286 e che vanno a leggere i descrittori si possono convertire più facilmente anche al 386.
A (**A**ccessed) è un bit che la CPU mette automaticamente a uno ogni volta il descrittore di quel segmento viene caricato in un registro di segmento. Ogni tanto il S.O. mette a zero questo bit in tutti i descrittori che gestisce. In questo modo esso è in grado di sapere, guardando i bit A che sono a uno, se il segmento è stato usato di recente e può basarsi su questa informazione per decidere quale segmento togliere dalla memoria durante una operazione di swap, oppure per capire quali segmenti sono stati influenzati da un'applicazione che è andata in crash.
Specificando il **TIPO** del segmento si indica nel relativo campo se il segmento contiene codice, dati od altro. La CPU può funzionare in modo leggermente diverso a seconda che essa utilizzi un segmento di codice od uno di dati. I segmenti di codice possono essere non leggibili e/o non scrivibili, mentre quelli di dati sono sempre leggibili e possono non essere scrivibili.
Il bit **S** determina se il segmento è di sistema o se contiene codice o dati. Se S = 0 il segmento è di sistema.
Il **D**escriptor **P**rivilege **L**evel (**DPL**) (due bit) indica a che livello di privilegio si deve essere per operare correttamente con il segmento (per il "privilegio", vedi dopo, in "protezione").
P (**P**resent) dice se il segmento considerato è attualmente presente in memoria fisica o meno (1 = presente). Se non è presente, la CPU lancia un'eccezione di "segment non present" (INT 11); il S.O. risponderà alla mancanza del segmento iniziando un'operazione di swap.
AVL (**av**ai**l**able = disponibile), un flag disponibile per l'uso da parte del software di sistema. Esistono istruzioni per attivarlo e per spegnerlo, di solito mantiene una informazione sul segmento utile al S.O.
Il bit **D/B** funziona in modo diverso in base al fatto che il segmento sia di codice o di dati. In ogni caso stabilisce se le operazioni di default in memoria debbono essere a 16 od a 32 bit.
0 è un bit riservato, non ancora usato dal sistema
G è il bit di "**g**ranularità", se G = 0 la granularità è "fine" ed i segmenti possono essere al massimo lunghi 1 MByte, se G = 1 la granularità è "larga" ed i segmenti possono arrivare a 4 GByte.

Determinazione dell'indirizzo fisico

La Figura 10 mostra la sequenza delle operazioni, quasi tutte "automatiche" eseguite dalla CPU quando deve fare la lettura di una locazione di memoria virtuale:

```
MOV AL, [2860A305965h]
```

L'indirizzo di inizio della tabella dei descrittori, viene ottenuto da GDTR o LDTR; nel punto (1) in Figura 10 si fa l'ipotesi che il segmento sia globale (TI = 0), per cui si usa GDTR.
Per puntare al descrittore del segmento voluto la CPU somma all'indirizzo di inizio della tabella il valore dei 13 bit di INDEX del selettore di segmento, moltiplicato per 8 (punto (3) e (4) in Figura 10); infatti, dato che ogni descrittore è lungo 8 byte, per giungere all'"INDEXesimo" descrittore bisogna aggiungere 3 zeri a destra, cioè moltiplicare per 8.
Se per errore il valore di INDEX fosse troppo grande ed appartenesse ad un segmento non allocato, la CPU sarebbe in grado di rilevare l'errore, perché il valore appena calcolato sarebbe maggiore dell'indirizzo dell'ultima locazione della tabella (calcolata come nel punto (2) di Figura 10).
All'interno del descrittore, il cui indirizzo è stato appena individuato, si trova l'indirizzo fisico dell'inizio del segmento (campo BASE), che vi era già stato scritto dal Sistema Operativo (punto (5) di Figura 10).
A questo punto per trovare l'indirizzo fisico della locazione di memoria virtuale cercata basta sommare all'indirizzo di inizio del segmento l'offset dell'indirizzo virtuale (punti (7) e (8) di Figura 10). A quell'indirizzo fisico l'istruzione farà accesso, in lettura o scrittura.
Qualora, per errore, la nostra MOV interferisse con altri task, il valore dell'indirizzo appena calcolato eccederebbe il valore di (indirizzo iniziale + LIMIT) indicato dal punto (6) di Figura 10, e verrebbe lanciata un'eccezione INT 13.

Granularità

I progettisti del 386, a corto di bit nel descrittore di segmento, si sono trovati a dover realizzare un compromesso nella definizione dei limiti dei segmenti.
I numeri che rappresentano la lunghezza del segmento (campo LIMIT, Figura 9) hanno solo 20 bit.

Il numero massimo rappresentabile con 20 bit è 1 MByte, per cui questa sarebbe la massima dimensione del segmento.
Per avere segmenti lunghi fino a 4 GByte si potrebbero aggiungere 12 bit a zero nella parte meno significativa di LIMIT, in modo che il numero diventi di 32 bit ma così il segmento di dimensioni più piccole sarebbe di 4 kByte ed aggiungere 1 al limite darebbe un segmento più lungo di 4 kByte.
Nel 386 è possibile scegliere, con il bit G del descrittore di segmento ("**granularità**"), fra due alternative per la lunghezza dei segmenti. G indica perciò la risoluzione con la quale si può definire la lunghezza di un segmento.
Se G = 0 la dimensione del segmento ha risoluzione di 1 byte ed esso può essere lungo da 1 byte a 1 MByte.
Se invece G = 1 la risoluzione è di 4 kByte ed esso può essere lungo da 4 kByte a 4 GByte.

Registri nascosti

Il meccanismo della memoria virtuale testè descritto prevede la lettura dalla memoria fisica degli 8 byte del descrittore ogni volta che deve fare un qualsiasi accesso in memoria.
Se i progettisti non avessero preso delle contromisure questo fatto rallenterebbe il processore in modo inaccettabile (8 trasferimenti dalla memoria invece di uno!).
Per questo è stata introdotta una parte "nascosta" dei registri di segmento, che si è già indicata nella Figura 6.
I registri di segmento contengono anche tutte le informazioni che sono memorizzate nel descrittore di segmento; esse vengono lette in memoria, dal descrittore, ogni volta che si carica un nuovo valore in un registro di segmento, e solo in quel momento. Quando si continua ad usare lo stesso segmento il descrittore verrà letto dal registro nascosto, all'interno della CPU, a non dalla memoria.
Ciò realizza un meccanismo veloce per la conversione da indirizzo virtuale ad indirizzo reale, per tutti gli indirizzi virtuali che stanno nello stesso segmento. I registri nascosti (detti anche "cache" dei descrittori di segmento) sono gestiti in modo completamente automatico dalla CPU, il programmatore non deve fare nulla al riguardo.

Per modificare il contenuto dei due registri GDTR e LDTR, fondamentali in questo meccanismo, sono state introdotte specifiche istruzioni, che naturalmente saranno le più "protette" istruzioni della CPU e potranno funzionare soltanto al livello di priorità più importante, quello riservato al nucleo del Sistema Operativo.
Al reset della CPU in GDTR e LDTR vengono scritti numeri fissi, stabiliti dal costruttore. I più importanti Sistemi Operativi per X86, che lavorano in flat mode, inizializzano al bootstrap[3] questi registri e non li cambiano mai.
Avendo studiato la segmentazione in modo protetto non si può non apprezzare come questo meccanismo, cervellotico e tutto sommato inutile in modo reale acquisti una valenza del tutto diversa se visto alla luce delle caratteristiche del modo protetto. In modo protetto "tutto torna" con la segmentazione, che diventa un meccanismo molto utile e potente, la cui complicazione non gioca a svantaggio del programmatore, ma gli rende disponibile un mondo intero di funzionalità avanzate e di possibilità creative.

Nel descrittore di segmento del 386 i campi sono spezzati in più parti per mantenere la compatibilità con quello del 286. Infatti nel descrittore di segmento del 286 gli ultimi due byte non furono utilizzati, dato che servivano meno informazioni per memorizzare il limite, che doveva essere di soli 16 bit e l'indirizzo iniziale, che prendeva solo 24 bit.

<FILE>
segmentaz.fh5
</FILE>
Figura 10: segmentazione X86: trasformazione da indirizzo virtuale ad indirizzo lineare

I numeri indicati in Figura 10 sono solo per esempio e non sono significativi in generale.

Alcune considerazioni sulla protezione della segmentazione

Usando la memoria segmentata la minima quantità di memoria che può essere "protetta" è un segmento.
Ciò significa che informazioni che appartengono a "categorie" diverse e che quindi devono essere protette in modo diverso devono stare in segmenti diversi.
Ciascun segmento ha, nel suo descrittore, oltre alle informazioni per l'accesso alla memoria fisica, informazioni sul livello di privilegio richiesto per poter lavorare su di esso (campo DPL). Ad ogni accesso alla memoria, sia in fase di fetch che in fase di execute, la CPU verificherà:
- che il programma in esecuzione abbia sufficiente privilegio per accedere alla locazione che ha richiesto
- che l'accesso sia all'interno del segmento in cui si lavora (che l'offset non sia maggiore di LIMIT).

Perciò per stabilire la protezione di un segmento bisogna sapere che privilegio bisogna avere per usarlo, quanto è lungo e di che tipo è il segmento utilizzato. Per questo queste informazioni sono incluse in tutti i descrittori di segmento.
Qualora un task tenti l'accesso ad una locazione di un segmento che ha DPL di valore minore di quello a cui la CPU sta funzionando, la CPU stessa genererà un'eccezione di errore di protezione nell'accesso in memoria (INT 13), cedendo in questo modo il controllo al Sistema Operativo, che risponde all'interruzione.
Anche la protezione delle istruzioni può generare un errore generale di protezione.

3 Vedi indice analitico della Parte 1

Per esempio, ogni tentativo di modificare il valore dei registri GDTR, LDTR con la CPU non in ring 0 provoca un'eccezione di protezione. In questo caso, al momento della generazione dell'eccezione, la CPU includerà nello stack un codice che identificherà l'errore di protezione avvenuto.

Protezione e salti
In ogni istante il livello di privilegio della CPU è quello associato al segmento di codice che sta eseguendo.
Eseguire una jump in un segmento diverso equivale perciò a cambiare il livello di privilegio. Per questo le jump devono essere istruzioni protette. Ogni volta che c'è un salto fra due segmenti di codice diversi la CPU verificherà che il suo livello di privilegio corrente sia migliore od uguale a quello del segmento di codice ove salta. Se questo non dovesse avvenire, la CPU genererà un'eccezione di protezione.
La protezione dalla modifica del codice funziona controllando il tipo dei segmenti. Non è possibile caricare il selettore ad un registro di segmento in un registro dati (DS, SS, ES, FS, GS). In CS ci può essere solo un selettore ad un segmento di tipo codice.
Se si deve passare ad eseguire codice di priorità superiore (p. es. da un applicativo ad una procedura del S.O.) se non si vuole avere un errore di protezione si deve usare quello che viene detto un "gate".
Un salto che implichi una variazione di privilegio non viene mai eseguito direttamente cambiando il contenuto di CS e IP, ma solo passando attraverso delle "tabelle di salti" (call gates), che sono gestite dal S.O. . Il S.O. è l'unico software che ha accesso alle tabelle dei gate e quindi ha il controllo su dove finiscono tutte le jump "pericolose" di tutti i programmi "normali". In questo modo non è possibile per un programma di livello utente cambiare privilegio senza che il S.O. lo sappia.
Un "CALL **gate**" è un descrittore in memoria, simile ad un descrittore di segmento, che contiene l'indirizzo di una procedura.
Un "gate" può essere innescato da: una jump, una chiamata a procedura, un interrupt, una trap o un "task gate" (che non spiegheremo).

Famoso è il gate A20 che è innescato se si tenta di avere accesso a locazioni oltre la fine del primo MByte di memoria fisica, cioè quando il ventunesimo piedino di indirizzo (A20 Address 20) viene attivato. Il piedino A20 non esisteva nel'8086; in quel caso, se si tentava si avere accesso a locazioni di indirizzo superiore a 1 MByte, si "ricominciava daccapo", senza segnalare altro, utilizzando locazioni di memoria di indirizzo basso. Alcuni software scritti (male) per l'8086 facevano uso di questa caratteristica di "andare a capo" nella memoria. Per assicurare la compatibilità di questo software, l'80286 introduce il gate A20, per identificare il momento in cui si ha l'overflow dell'indirizzo in memoria e far partire un software che emuli il comportamento dell'8086.

Anche i port di I/O hanno un descrittore che indica il livello di privilegio richiesto per il loro uso, così che le istruzioni IN o OUT su quei port siano possibili solo quando il livello di privilegio di I/O della CPU è sufficientemente buono.
L'indicazione del livello di privilegio di I/O al quale la CPU sta lavorando in ogni istante è mantenuta in due bit detti **IOPL** (**I**nput **O**utput current **P**rivilege **L**evel), nel registro dei flag.

Memoria virtuale paginata

Come avevamo accennato precedentemente, dal 386 in poi è possibile avere due meccanismi per la memoria virtuale: la memoria virtuale segmentata e quella paginata.
Fino ad ora abbiamo trascurato la memoria virtuale paginata, fingendo che fosse disabilitata. In effetti nelle CPU X86 è possibile disabilitare la memoria virtuale paginata[4]; quando ciò accade tutto funziona esattamente come abbiamo già descritto.
Qualora invece la paginazione sia abilitata l'indirizzo "prodotto" dalla segmentazione non è più un indirizzo fisico, ma un altro indirizzo virtuale, da 32 bit.
L'indirizzo che esce dal meccanismo della segmentazione viene detto "**indirizzo lineare**". L'indirizzo lineare è di 32 bit è può essere o meno un indirizzo virtuale.
Se la paginazione non è abilitata, l'indirizzo lineare a 32 bit, ottenuto con la segmentazione, è usato come indirizzo fisico.
Se invece la paginazione è abilitata l'indirizzo lineare non è fisico, ma virtuale. Per tradurre questo ulteriore indirizzo virtuale a 32 bit in un indirizzo fisico, pure a 32 bit, si fa riferimento a tabelle, in modo analogo in linea di principio quanto fatto per la segmentazione. Peraltro, come vedremo le tabelle sono molto diverse.

Il meccanismo della **paginazione** (paging) realizza un secondo livello di virtualizzazione della memoria, dopo la segmentazione, più efficiente e semplice dal punto di vista della gestione da parte del S.O..

La memoria virtuale paginata gestisce blocchi di lunghezza fissa detti "pagine". Negli X86 è uno spazio di indirizzi di 4 GByte diviso in pagine della lunghezza di 4 kByte.
In ogni istante ognuna di queste pagine può risiedere in memoria fisica, ed allora sarà utilizzata senza problemi da parte della CPU, oppure può essere "parcheggiata" sull'hard disk, ed allora verrà lanciata l'eccezione di "page fault" (INT 14), ed il S.O. farà in modo, effettuando uno swap, che la pagina mancante torni in memoria fisica.
Le pagine in memoria fisica sono una di seguito all'altra; quindi non possono iniziare ad indirizzi qualsiasi, ma solo ogni 4 kByte. Questo vuol dire che l'indirizzo di inizio di una pagina avrà gli ultimi 12 bit a zero.

4 Per farlo bisogna scrivere 0 nel bit PG (paging) del registro CR0

L'indirizzo lineare a 32 bit può essere suddiviso in tre "campi", detti "**DIR**", "**PAGE**" e "**OFFSET**". DIR e PAGE prendono 10 bit ciascuno, OFFSET 12 bit (totale 32!).

<FILE>
IndLin386.FH5
</FILE>

Figura 11: indirizzo lineare virtuale, suddiviso in campi per la paginazione

Per ottenere un indirizzo fisico è necessario passare attraverso due tabelle chiamate **page directory** e **page table**, che sono normali pagine di 4 kByte come le altre.
Queste tabelle contengono "descrittori di pagina", che hanno tutti lo stesso formato (vedi Figura 13) ed occupano ciascuno 32 bit (4 Byte). Si noti che la dimensione dei descrittori di pagina è la metà di quella dei descrittori di segmento.
Ogni tabella di descrittori di pagina ha la lunghezza di una normale pagina, cioè 4kByte; perciò può contenere fino a 1k descrittori (1024 = tabella di 4kByte / 4 byte per descrittore).

Per contenere l'indirizzo fisico (a 32 bit) della prima di queste due tabelle la CPU utilizza i 20 bit più significativi di un nuovo registro della CPU, che ha nome **CR3**. A questi 20 bit aggiunge 12 zero a destra (moltiplica per 4k), puntando così all'indirizzo fisico a 32 bit di inizio della prima tabella delle pagine, la "page directory" (punto (1) di Figura 12).
Dato che i descrittori all'interno della tabella sono 1k servono solo 10 bit per ottenere, noto l'indirizzo iniziale, l'indirizzo di un qualsiasi descrittore. I primi 10 bit dell'indirizzo virtuale lineare servono a puntare ad uno specifico descrittore dentro la "page directory" (punto (2) di Figura 12).
Per far questo la CPU aggiunge due zeri a destra del numero DIR (moltiplica per 4), che da 10 diventa di 12 bit (punto (3) di Figura 12), e somma questi 12 bit all'indirizzo fisico a 32 bit (punto 4).
Questo ci dà l'indirizzo fisico del descrittore dentro la tabella "page directory" (punto 5).
Il descrittore contiene 32 bit, dei quali 20 servono per individuare il primo byte della seconda tabella ("page table") (punto 6 di Figura 12); gli altri 12, come vedremo, servono per contenere informazioni varie sulla pagina.
I 20 bit più significativi del descrittore vengono trasformati in un numero a 32 bit, aggiungendo 12 zeri a destra (moltiplicando per 4 k) (punto 7 di Figura 12). Ciò dà l'indirizzo fisico di inizio della page table (punto 8).
Per avere l'indirizzo del secondo descrittore, la CPU usa la parte PAGE dell'indirizzo virtuale (punto 9), aggiungendo 2 zeri a destra (punto 10), analogamente a quanto visto prima. Poi lo somma con l'indirizzo a 32 bit ottenuto prima (punto 11). Questo produce l'indirizzo del secondo descrittore (punto 12), che ci manderà all'inizio della pagina cercata, quella dove ci sono i dati che cerchiamo (punto 15).

<FILE>
paginazione.FH5
</FILE>
Figura 12: paginazione 386

Per avere l'indirizzo di inizio della pagina il procedimento è identico al precedente. Il numero che si ottiene dal descrittore è di 20 bit (punto 13 di Figura 12), esso va "aumentato" fino a 32 bit aggiungendo 12 zeri (punto 14). Ciò produce l'indirizzo di inizio della pagina (punto 15).
Per ottenere (finalmente) l'indirizzo fisico su cui lavorare (punto 18), basta aggiungere all'inizio della pagina l'offset a 12 bit, rappresentato dall'ultima parte dell'indirizzo lineare (punto 17 di Figura 12), quella che non avevamo ancora usato (punto 16).
Da notare che l'offset finale non viene moltiplicato per nessun fattore, essendo già il numero a 12 bit. Esso ci deve permettere, avendo l'inizio della pagina, di andare a prendere uno qualsiasi dei byte della pagina di 4 k.

La relazione fra i segmenti e le pagine è abbastanza "lasca", ciò rende molto flessibile l'utilizzazione congiunta dei due meccanismi. Un segmento, che può essere lungo da un byte a 4 GByte, può occupare più di una pagina; mentre in una pagina possono stare anche più di un segmento.

Descrittori di pagina

Le pagine hanno lunghezza fissa. Non è quindi necessario memorizzarne la dimensione nei descrittori di pagina. Inoltre, visto che la pagina è lunga 4 k, non è necessario memorizzare tutto l'indirizzo iniziale, ma i soli 20 bit più significativi, visto che comunque l'inizio della pagina avrà tutti gli ultimi 12 bit meno significativi a zero.
Questi due fattori portano il descrittore di pagina ad essere molto più piccolo del descrittore di segmento, e ad occupare solo 32 bit. La struttura del descrittore di pagina è la seguente:

<FILE>
DescPag386.FH5
</FILE>

Figura 13: descrittore di pagina

Come già visto, il descrittore di pagina contiene, nei suoi 20 bit più significativi, i 20 bit più significativi dell'inizio della pagina cui punta. Gli altri bit del descrittore della page table contengono informazioni su protezione ed accesso alla pagina ed hanno il seguente significato:

P (bit 0) **P**resent: se P = 0 vuol dire che la pagina non è presente in memoria fisica, la CPU inizia un'eccezione di "page fault"
R/W (bit 1) **R**ead / **W**rite: se = 1 la pagina si può leggere e scrivere, altrimenti si può solo leggere
U/S (bit 2) **U**ser / **S**upervisor: serve per la protezione della pagina, che ha solo due livelli: "user" e "supervisor"
A (bit 5) **A**ccessed: viene messo a 1 quando si fa un accesso, in lettura o scrittura, ad una locazione della pagina
D (bit 6) **D**irty: viene messo a 1 quando si fa un accesso in scrittura ad una locazione della pagina
AVL (bit 9-11) **Av**ai**l**able: tre bit lasciati a disposizione del programmatore del Sistema Operativo

I bit 4 e 5 sono usati solo dal Pentium in poi, per indicare come si deve gestire la memorizzazione delle pagine nelle cache interne:

PWT (bit 3) **p**age cache **w**rite **t**hrough: abilita il funzionamento write through del caching della pagina
PCD (bit 4) **p**age **c**ache **d**isable: disabilita il caching di questa pagina

Il descrittore contenuto nella prima tabella (page directory) è leggermente diverso da questo.

La funzione dei bit P, A e D è analoga a quella descritta per la segmentazione. I bit A e D sono automaticamente messi a uno dalla CPU, quando si fa un accesso alla pagina, ma non vengono mai cancellati automaticamente, ciò deve essere fatto esplicitamente da un programma (di solito è il S.O. che lo fa).
I bit AVL possono essere letti e scritti dai programmi di sistema e vengono usati dai S.O. per memorizzare informazioni addizionali per la protezione e la sicurezza.

Modello piatto

Dato che con il solo offset è possibile coprire uno spazio di indirizzi di 4 GByte, più che sufficiente per la maggioranza delle applicazioni odierne, risulta interessante far in modo che la segmentazione, con la sua complicazione, sia di fatto esclusa dal funzionamento dei programmi e dei Sistemi Operativi.
Ciò si può realizzare facendo puntare ogni registro di segmento allo stesso descrittore, e quindi allo stesso segmento (vedi Figura 14).
Questo unico descrittore di segmento viene inizializzato in modo da coprire tutto lo spazio di indirizzamento lineare, mettendo 0 nell'indirizzo BASE, il valore massimo (FFFFFh) come limite di lunghezza ed impostando la granularità "larga" (bit G = 1).
In questo modo codice, dati e stack stanno tutti in un unico grande spazio d'indirizzi di 4 GByte.

<FILE>
Flat.FH5
</FILE>

Figura 14: modello piatto

Questo modo di inizializzare in segmenti viene detto "**modello piatto**" ("flat model"), proprio perché si ha la comodità di uno spazio continuo di indirizzi di 4 GByte.
Naturalmente questi indirizzi possono essere virtuali, se la paginazione è abilitata, o fisici, se non lo è.
In modo "flat" i registri di segmento vengono sistemati alla partenza del computer e non vengono più toccati. In questo modello un identico offset significa un identico indirizzo lineare.
Siccome non è pratico cambiare il modello di memoria virtuale durante il funzionamento del computer esso rimane lo stesso in ogni Sistema Operativo.
I progettisti dei Sistemi Operativi devono quindi decidere sin dall'inizio se realizzarlo con memoria virtuale segmentata, segmentata e paginata o piatta, ovverosia solo paginata.
Data la "vastità" dello spazio virtuale di 4 GByte e la semplicità del modello piatto, i sistemi operativi attuali per X86 a 32 bit utilizzano la CPU in flat mode (es. tutti gli Unix e tutti i Windows).

Protezione con la paginazione
Come si vede analizzando il contenuto del descrittore di pagina (Figura 13), usando la paginazione non ci sono quattro livelli di protezione, come nella segmentazione, ma solo due. I task che usano le pagine possono perciò essere in modo "supervisor", che corrisponde ai livelli di privilegio 0, 1 e 2 della segmentazione (ring 0, 1 e 2), oppure in modo "user", che corrisponde a ring 3.
Quando la CPU esegue in modo supervisore può accedere a tutte le pagine, mentre se è in modo utente può accedere solo alle pagine che sono definite a livello utente nel loro descrittore. La protezione delle pagine si applica anche definendole di sola lettura o di lettura/scrittura con il bit R/W del descrittore.
Quando la paginazione è attivata la CPU valuta prima la protezione dovuta alla segmentazione poi quella dovuta alla paginazione.
Se si programma in modo "flat" (tutti i selettori di segmento a 0 ed un solo segmento di 4 GByte) si perdono i benefici della protezione hardware a quattro livelli. I Sistemi Operativi che funzionano in questo modo dovranno realizzare la

protezione con la sola paginazione, che ha solo due livelli di priorità, e con l'utilizzazione dei tre bit "available" del descrittore di pagina.

1.3.3 Interrupt ed eccezioni in modo protetto

Se una CPU X86 funziona in modo protetto la tabella dei vettori d'interruzione può risiedere in qualsiasi punto della memoria fisica.

La tabella dei vettori ha struttura e nome nuovi; si chiama IDT (**I**nterrupt **D**escriptor **T**able) ed è una tabella di descrittori, di contenuto analogo a GTD o LDT.
Per sapere dove risiede la tabella si usa il registro IDTR (**I**nterrupt **D**escriptor **T**able **R**egister), che punta alla locazione in memoria fisica in cui risiede la IDT. Per caricare e scaricare il contenuto di questo registro esistono le istruzioni LIDT e SIDT.
Il numero dei possibili vettori d'interruzione rimane 256. Il numero dell'interrupt hardware che deve essere eseguito viene letto dal data bus, come nell'8086.
A differenza dell'8086, ove la tabella dei vettori conteneva soltanto 4 Byte per vettore d'interruzione (IP e CS della ISR), nel 286 essa contiene un descrittore di 8 Byte per ogni vettore, analogo ad un descrittore di segmento, e può anche non contenere tutti e 256 i vettori possibili (per questo il registro IDTR contiene anche la lunghezza della tabella).
I descrittori contenuti nella IDT possono essere di tre tipi; in base al tipo di descrittore il comportamento della CPU quando riceve l'interrupt può essere diverso.
Il tipo più comune di interrupt è l'"interrupt gate". Il suo descrittore contiene, fra le altre cose, un indirizzo virtuale segmentato completo, di 46 bit. Questo è l'indirizzo virtuale della ISR.
Attraverso il meccanismo, già descritto, della memoria virtuale segmentata la CPU si procura, avendo l'indirizzo virtuale, l'indirizzo fisico della ISR. Il processo è illustrato dalla Figura 15.

<FILE>
Int386.FH5
</FILE>

Figura 15: interrupt X86 in modo protetto (386 >)

Come nell'8086 la CPU salva nello stack tutti i flag, CS del programma interrotto, il suo offset a 32 bit e, a differenza dell'8086 e solo in alcuni casi, un codice di errore, che identifica, per esempio, il tipo di eccezione lanciata.
Poi salta alla ISR il cui indirizzo è stato determinato nel modo descritto.
L'esecuzione di un interrupt può determinare o meno la variazione del livello di privilegio, in base al tipo di descrittore contenuto nella IDT. L'interrupt può anche essere programmato per generare automaticamente un task switch completo.

Eccezioni

È ormai chiaro che il lancio un un'eccezione è il meccanismo di elezione quando la CPU deve comunicare al S.O. una qualsiasi condizione anomala, risolvibile o meno.
Per questo nelle CPU moderne, che debbono supportare al meglio i S.O. multitasking, sono state aggiunte un gran numero di nuove eccezioni.
Per quanto riguarda il salto alla ISR, le eccezioni sono del tutto analoghe agli interrupt hardware e ne condividono la IDT.
Nell'Appendice A1 sono indicate tutte le eccezioni che sono presenti in un X86 della classe Pentium.
Una cosa importante, da notare in questa tabella, è che i numeri di INT di alcune eccezioni sono gli stessi di quelli di alcune IRQ hardware dell'architettura del PC. L'IBM scelse di utilizzare quei numeri senza sapere < : ^(che sarebbero stati utilizzati per scopi interni in CPU successive all'8086. Il PIC master di un PC emette interrupt che hanno numero da 8 a 17, perciò l'eccezione 13 (general protection, molto utilizzata), avrebbe lo stesso numero di INT della porta parallela LPT2. Questo vuol dire che il PIC deve essere riprogrammato perché emetta numeri di INT diversi quando è in modo protetto rispetto a quelli, "standard", in modo reale.

1.3.4 Supporto della CPU al cambio di task (task switching)

Per un X86 un cambio di task (task switch) è simile ad una chiamata a procedura ma deve memorizzare molte informazioni in più. Infatti per una procedura basta il salvataggio del solo indirizzo di ritorno, mentre per un cambio di task è necessario salvare tutti i registri ed anche altre informazioni.
Inoltre un task non "ritorna" come una procedura per cui il salvataggio non può essere fatto nello stack, ma in una struttura dati apposita, situata in memoria.
La struttura dati usata dagli X86 è il TSS (**T**ask **S**tate **S**egment). Il TSS è un particolare tipo di segmento, che ha il suo codice nel campo "TIPO" ed anche un particolare tipo di descrittore (TD = **T**ask **D**escriptor).
La struttura del TSS di un AMD K5 (compatibile X86, la prima parte del TSS è specifica delle CPU AMD, il descrittore compatibile Intel parte da 64h) è illustrata nella Figura 16.

!!!! mettere

Figura 16: TSS (dal manuale "AMD-K5 Processor Development Guide")

Il TSS contiene, fra l'altro: EIP, tutti gli altri i registri generali e di segmento, tutti i flag (EFLAG), l'indirizzo della LDT (che può essere caricato in LDTR), CR3 (per la paginazione), tre puntatori di stack, informazioni per l'accesso protetto ai port di I/O.
Nel caso perciò che un task debba essere interrotto, e debba quindi perdere il controllo della CPU (preemption), la CPU stessa eseguirà una sequenza automatica di operazioni, che porteranno alla scrittura di tutto il descrittore del task, cioè al "congelamento" in memoria della situazione attuale di tutti i registri. In questo modo, quando il task potrà assumere nuovamente il controllo della CPU, esso potrà ripartire esattamente dalla stessa condizione nella quale era rimasto precedentemente.
L'indirizzo del TSS del task corrente è puntato in memoria fisica da un registro della CPU detto TR (**T**ask **R**egister). Esistono due istruzioni specifiche per caricare e scaricare il contenuto del TR (LTR ed STR).

1.3.5 Modalità V86

La differenza è che la memoria utilizzata non è la memoria fisica ma il primo MByte del segmento di memoria virtuale utilizzato dal task.
La cooperazione fra CPU che lavora in modalità V86 ed il Sistema Operativo realizza una vera e propria macchina virtuale ben supportata dall'hardware, alla quale il S.O. può assegnare indirizzi virtuali ben determinati e che può essere "fermata in tempo" se tenta di aver accesso a locazioni non permesse.
In questo modo viene facilitato il compito di far eseguire programmi scritti per l'8086, ma viene salvaguardato il funzionamento degli altri task in un sistema multitasking. E' inoltre possibile eseguire "contemporaneamente" più di una macchina virtuale 8086, e quindi più di una copia del sistema operativo DOS, ciascuna delle quali esegue un programma DOS.
In Win (anche NT) e Linux per 386 metà dello spazio d'indirizzi virtuale (da 80000000h a FFFFFFFFh) è utilizzato come memoria condivisa fra i processi ed il S.O. . La memoria virtuale "alta" può perciò essere visibile da ogni processo e dal S.O.. La metà "bassa" della memoria virtuale viene "divisa" fra i processi. Il S.O. la alloca ai processi che la usano in modo esclusivo.

1.4 Memoria virtuale in Linux

Riferimenti:

Intel Corporation "Pentium ® Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual"

Advanced Micro Devices (AMD) "AMD-K5 ™ PROCESSOR Software Development Guide"