Gestione della memoria e dei processi in Linux su 80x86

 Elias S. G. Carotti <ecarotti@athena.polito.it>
Enrico Masala <masala@athena.polito.it>

The Linux Documentation Project

Disclaimer

Gli autori declinano ogni responsabilità per le informazioni (e per il loro eventuale utilizzo) contenute in questo documento e non garantiscono neanche la rispondenza a quanto c'è nei sorgenti del kernel, per quanto non dovrebbero esserci grossi errori, dato che gli autori hanno fatto tutti gli sforzi per accertarsi dell'accuratezza del documento.

Nessuno è da ritenersi responsabile per eventuali danni dovuti all'utilizzo delle informazioni.

Nel caso si trovassero degli errori, si prega di inviare correzioni e precisazioni agli autori, che sono contattabili agli indirizzi:

ecarotti@athena.polito.it
masala@athena.polito.it

Indice

1 Prefazione **5**

2 Introduzione **7**

3 Organizzazione della memoria **11**

3.1 Architettura di base 11

3.2 Segmentazione 12

3.3 Livelli di privilegio 14

3.4 Gli interrupt 14

3.5 System calls 15

4 La paginazione **19**

4.1 Page allocation 21

4.2 L'organizzazione della memoria di un processo 22

4.3 Il page fault 24

4.4 On–demand–loading 26

4.5 Page cache 27

4.6 Swapping 28

4.7 Gestione del bug del pentium 30

5 Context switching **33**

5.1 La routine 34

5.1.1 Problema del coprocessore 35

6 Gestione dei processi **37**

6.1 La `fork()` 38

6.2 La `exec()` 39

7 Considerazioni generali **41**

8 Variazioni con il kernel 2.2 **45**

8.1 Segmentazione 45

8.2 Context switching 46

9 Riferimenti **47**

A L'assembler AT&T **49**

A.1 Le estensioni CYGNUS all'assembler inline 49

B Glossario **53**

Elenco delle figure

3.1 Organizzazione della memoria 12
3.2 Organizzazione della **GDT** 13
3.3 Gli interrupt 15
3.4 Utilizzo dei GATE 16

4.1 Le page table 19
4.2 Kernel Page Tables 20
4.3 Gestione della memoria 22
4.4 Memoria di un processo 23
4.5 Copy on write 25
4.6 On–demand–loading 27
4.7 Protezione della IDT 32

7.1 Segmentazione e paginazione 41
7.2 Indirizzamento 43

Capitolo 1

Prefazione

Lo scopo fondamentale di questo lavoro è di analizzare come è stato organizzata la gestione dei processi e della memoria nel sistema operativo Linux, e come tutto ciò sia stato implementato sulle architetture della famiglia x86.

Lo studio è stato condotto visionando i sorgenti del kernel 2.0, con alcuni cenni a quelli della versione 2.2.

La lettura presuppone di avere già qualche conoscenza del funzionamento interno dei sistemi operativi, in particolare di tipo Unix.

Si inizia naturalmente dalla gestione della memoria, in quanto è fondamentale sapere come questa venga suddivisa ed organizzata, poich ciò incide notevolmente sulle prestazioni e sul buon funzionamento di tutto il sistema, ed è inoltre un requisito indispensabile per poter comprendere quale sia l'organizzazione dei processi nel sistema.

Si analizza in particolare il meccanismo di paginazione e di segmentazione, sia nella parte indipendente dall'architettura e quindi propria del sistema operativo Linux, sia in quella che la realizza sulla particolare architettura x86. In quest'ultima si è posto in evidenza come i principali meccanismi per la paginazione e la segmentazione sono stati impostati, sia per quanto riguarda il processore, sia per le strutture da quest'ultimo usate, ma poste in memoria.

Si discute quindi il ruolo del page fault nel sistema e i compiti che vengono svolti dal relativo handler. E' molto importante capirne il funzionamento, in quanto questa routine è alla base dell'intera gestione della memoria: in essa vengono realizzati, per esempio, il copy on write e l'on demand loading, nonch viene riconosciuta la necessità di operare lo swapping delle pagine.

Il *copy–on–write* è una filosofia di gestione delle pagine tramite la quale si cerca di

rendere il massimo numero di quelle comuni a più processi, al fine di massimizzarne l'uso. Il meccanismo consente di condividere anche le pagine contenenti dei dati modificabili, duplicandole effettivamente solamente nel momento in cui queste diventino differenti: di conseguenza il numero di pagine utilizzate dal sistema è sempre il minimo, con gli ovvi vantaggi che ciò comporta.

L'*on–demand–loading* consente invece di caricare le pagine dell'eseguibile solo nel momento in cui queste devono essere eseguite dal programma, ottenendo quindi un risparmio di memoria effettivamente utilizzata.

Si illustra infine come la routine del page fault ponga inoltre rimedio al noto bug F00F del pentium, impedendo che un'istruzione non privilegiata blocchi completamente il sistema, compromettendone cosìla sicurezza.

Naturalmente non si può evitare di descrivere come avvenga il context switching: nel kernel 2.0 ciò è realizzato tramite il supporto dei meccanismi hardware dei processori x86.

Si commentano inoltre le istruzioni assembler che lo implementano, le quali non sono facilmente comprensibili a prima vista. Dovendo parlare dei processi, è d'obbligo descrivere due meccanismi fondamentali del sistema operativo: la creazione di nuovi task e la loro esecuzione.

Per fare ciò si è analizzato il codice delle system call `fork()` ed `exec()`, mettendo in evidenza quali modifiche esse apportino alle tabelle di gestione del sistema, ed in particolar modo a quelle della memoria, per assolvere al loro compito.

Si pone in evidenza come il funzionamento di queste azioni sia strettamente legato al page fault: quest'ultimo, in particolare tramite il meccanismo del copy on write, permette di ottenere un'implementazione particolarmente efficiente di queste chiamate di sistema.

Un capitolo è inoltre dedicato a delle considerazioni generali sul sistema, quali per esempio la scelta dell'indirizzo base dei segmenti del kernel, che nei sistemi x86 è fonte di alcune limitazioni, in parte dovute all'organizzazione data al sistema operativo, ed in parte dovute all'hardware utilizzato.

Infine si descrivono alcune tra le più importanti variazioni incluse nella versione 2.2 del kernel, tra le quali la modifica della segmentazione e del context switching, ora implementato tramite routine software.

In chiusura una breve appendice guida nella comprensione dell'assembler scritto in standard AT&T, spesso utilizzato nei sorgenti di Linux, e delle estensioni del gcc all'assembler inline.

Capitolo 2

Introduzione

Linux è un sistema operativo molto esteso: si pensi che solo l'ultima versione compattata del kernel, che contiene solo i sorgenti e una minima parte di documentazione, è costituita da ben 10 MB di dati.

Ovviamente in questa descrizione ci si occuperà solo di una parte del kernel, in particolare di quella che gestisce la memoria e i processi nel sistema, che è senza dubbio piuttosto estesa.

Linux è un sistema che è stato inizialmente sviluppato sull'architettura 80386 della Intel. Oggigiorno la parte principale di gestione del sistema è indipendente dall'architettura, grazie allo sforzo compiuto da svariati sviluppatori in ogni parte del mondo. Restano cosìdipendenti dal sistema solo alcune parti del codice, tra cui per esempio la gestione in dettaglio della paginazione, della segmentazione, del processo di boot e del context switching.

Nella presente trattazione si è cercato di mettere in luce sia i meccanismi di alto livello che presiedono al funzionamento del kernel, sia alcuni dettagli implementativi dello stesso sull'architettura x86.

Al fine di poter seguire la presente descrizione, è necessario prima di tutto sapersi orientare tra le varie versioni. Esse sono numerate tramite un identificatore costituito da tre cifre, separate tra di loro da dei punti. Il primo numero è il major version number (le versioni sviluppate finora appartengono alla serie 0, 1 o 2), e viene incrementato generalmente solo in caso di modifiche sostanziali al kernel. Il secondo numero è il minor version number, che può essere pari o dispari: nel primo caso indica che si tratta di una versione giudicata stabile, mentre il secondo è la versione che viene periodicamente modificata dagli sviluppatori, ed è quindi più sperimentale e di conseguenza, non essendo testata completamente, più instabile. Il terzo numero è utilizzato per indicare il livello di "patch" corrente: la "patch" è una

modifica al kernel che in genere aggiunge nuove funzionalità o corregge i bachi riscontrati.

Fatta questa premessa, si può ora spiegare che questa descrizione si basa principalmente sulle versioni del kernel 2.0.33 e successive (quindi giudicate stabili), in quanto erano le più recenti disponibili al momento dell'inizio di questo lavoro. Da non molto è uscita la versione 2.2.0, le cui principali modifiche apportate sono state descritte al termine della presente trattazione.

Innanzitutto si noti che in questa descrizione, quando si fa riferimento ad un file dei sorgenti, il relativo pathname è stato scritto in modo relativo alla directory principale dei sorgenti (che usualmente sono posizionati in /usr/src/linux).

La suddivisione dei sorgenti di Linux nelle varie sottodirectory segue questo schema generale, indipendente dalle varie versioni:

kernel		parte relativa alla gestione generale del sistema
mm		parte della gestione della memoria
net		parte di gestione dei protocolli di rete
fs		parte di gestione del file system
drivers		parte dei drivers specifici dei periferici
arch		parte contenente codice dipendente dall'architettura (es. i386)
	boot	parte di basso livello per l'inizializzazione del sistema
	kernel	parte del kernel dipendente dall'architettura
	mm	parte del memory management dipendente dall'architettura
	lib	parte contenente funzioni di utilità
	math-emu	parte contenente il codice di emulazione del coprocessore
include		file di include per i sorgenti, suddivisi in due parti
	linux	file generali di Linux
	asm	file dipendenti dall'architettura (es. asm-i386)
init		parte di alto livello per l'inizializzazione del sistema
ipc		parte della gestione dell'Inter Process Communication
lib		parte comprendente alcune funzioni di utilità
modules		parte di gestione dei moduli caricabili a run-time nel kernel
scripts		file di shell per la configurazione, ad esempio tramite menu

Inoltre è presente una directory Documentation che contiene diverse spiegazioni sulle

operazioni basilari per far funzionare il kernel.

Per una agevole lettura è consigliabile avere alcune conoscenze del funzionamento e dell'organizzazione interna dei sistemi operativi di tipo Unix, ed inoltre può aiutare una sommaria conoscenza del funzionamento dei meccanismi di protezione, di segmentazione e paginazione dei processori, in particolare della famiglia x86 in modo protetto.

Se ci si vuole addentrare nei sorgenti, è fondamentale una buona conoscenza del linguaggio C, in quanto questi sono spesso scritti in maniera non facilmente comprensibile a prima vista, causata sovente dal desiderio di ottenere una buona ottimizzazione da parte del compilatore.

Infine è utile possedere qualche rudimento di linguaggio assembler, in particolar modo di quello AT&T, che differisce un po' dal consueto linguaggio Intel: ciò è particolarmente utile per visionare le routine di basso livello, quali quelle di inizializzazione delle tabelle di segmentazione e paginazione, di impostazione dei vari bit nei registri del processore, nonch il codice del context switching.

Capitolo 3

Organizzazione della memoria

Non vi è dubbio che l'organizzazione della memoria sia un aspetto fondamentale per le prestazioni dell'intero sistema operativo.

L'implementazione che ne deriva è dunque piuttosto complessa ed estesa, anche a causa del fatto che un sistema operativo moderno richiede numerose funzionalità. Tra queste ricordiamo: la memoria virtuale, che consente di avere processi di dimensione totale superiore alla memoria fisica; l'allocazione e deallocazione efficiente della stessa ai programmi, che implica meccanismi per ridurre la frammentazione; la cache del file system, meglio se di dimensioni variabili secondo la memoria disponibile di volta in volta.

Per rendersi conto della complessità è sufficiente notare la quantità dei sorgenti e la dimensione dei file che si trovano nella directory mm.

3.1 Architettura di base

E' noto che per realizzare la memoria virtuale vi sono diverse possibili strategie implementative, basate sull'uso della segmentazione e della paginazione. Gli sviluppatori di Linux, per la versione 2.0 che abbiamo esaminato, hanno scelto di usare il meccanismo della segmentazione paginata.

E' noto che con questo sistema l'indirizzo virtuale specificato dal programma deve prima attraversare un processo di segmentazione e poi uno di paginazione, per poter essere tradotto in un'indirizzo fisico utilizzabile per accedere alle celle di memoria. Nella particolare architettura qui descritta, ossia quella della famiglia x86, è possibile avere il supporto dell'hardware per entrambe le operazioni precedentemente descritte, ottenendo quindi una certa efficienza.

Scendendo più in dettaglio, si può dire che la segmentazione è utilizzata da Linux su questa architettura più per ragioni di implementazione delle protezioni che per una suddivisione dei processi in parti omogenee. Quest'ultima (ossia la divisione dei programmi in parte codice, parte dati, librerie condivise ecc..) è invece ottenuta solamente per via software, e quindi in maniera completamente indipendente dall'architettura. Si vedrà più avanti come in questa organizzazione la struttura `vm_area_struct` giochi un ruolo fondamentale, in quanto è utilizzata per caratterizzare le diverse porzioni di memoria dei programmi.

3.2 Segmentazione

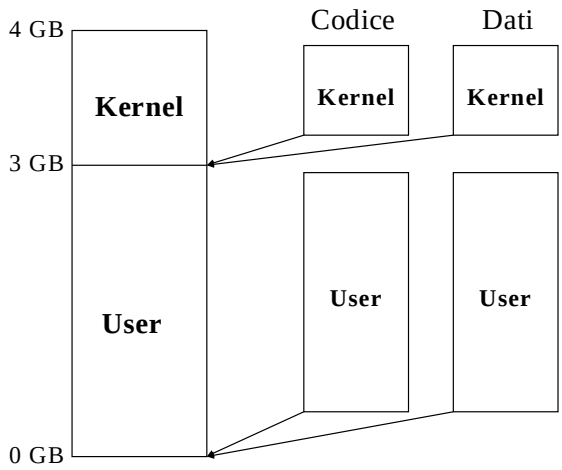

Figura 3.1: Organizzazione della memoria

I segmenti in Linux sono definiti come si vede dalla figura 3.1.

Il codice di inizializzazione si trova nel file arch/i386/kernel/head.S. Essi vengono resi operativi nel momento del passaggio del processore dal modo reale al modo protetto, istante nel quale si incominciano ad utilizzare i selettori per l'accesso alla memoria.

Poich essi riguardano tutto il sistema e sono indipendenti dal particolare processo che si sta eseguendo, sono ovviamente posti nella Global Descriptor Table (GDT). Si riporta qui di seguito l'organizzazione di detta tabella, dove ogni riga simboleggia un descrittore di 8 byte: Si notano le righe corrispondenti ai 4 segmenti citati, due per il modo user con DPL (Descriptor Privilege Level) pari a 3, e due per il modo kernel con DPL pari a 0. Sono presenti due righe per ogni modalità in quanto una è utilizzata per gli accessi necessari al caricamento del codice, mentre l'altra è usata per manipolare i dati (ed ha quindi permessi sia di lettura sia di scrittura a differenza della precedente). Inoltre la tabella contiene anche una coppia di descrittori per ogni processo che può girare sul sistema: esse si riferiscono

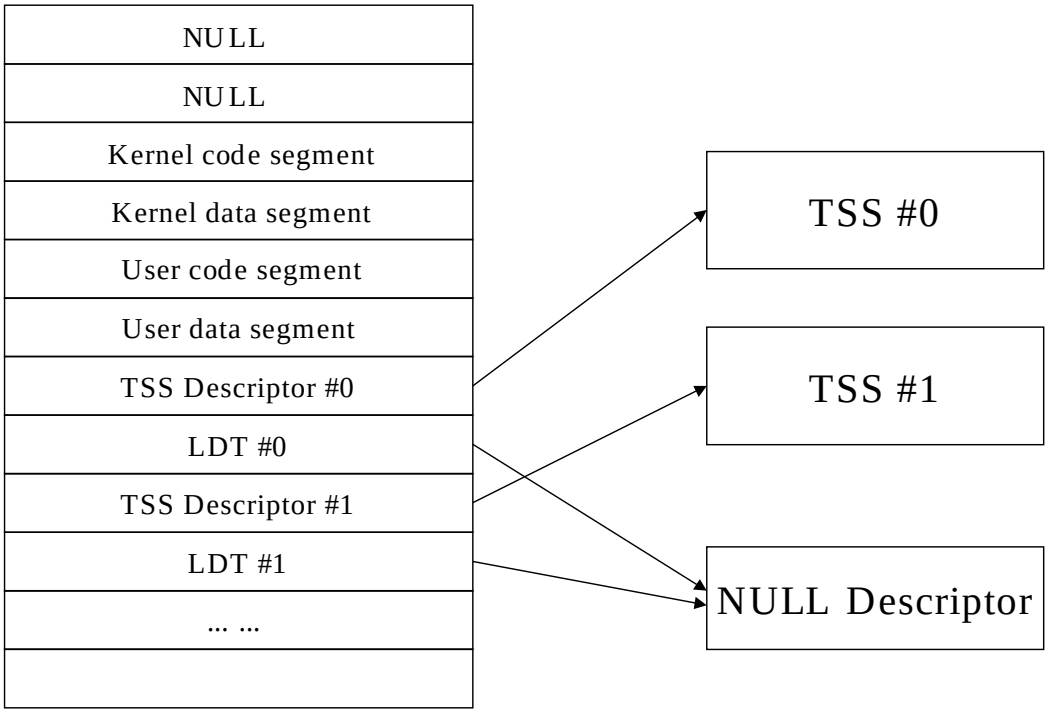

Figura 3.2: Organizzazione della **GDT**

rispettivamente alla Local DescriptorTable (LDT) e al Task State Segment (TSS) di ciascun task. Il posizionamento di questi descrittori nella **GDT** impone dunque che il massimo numero dei processi su architettura x86 sia pari a poco più di 4000, poich la **GDT** è limitata a 8192 entry; il valore di default è comunque di 512 processi (modificabile tramite una direttiva `#define NR_TASKS 512` nel file include/linux/tasks.h). Sono infine presenti alcuni segmenti utilizzati per la gestione delle funzionalità di power saving dei processori x86.

Osservando meglio la figura 3.2, si nota come l'entry relativa alla **LDT** punti normalmente sempre alla stessa struttura dati che contiene un descrittore nullo. Questo è dovuto al fatto che Linux non prevede di utilizzare in maniera sistematica la **LDT** per la gestione dei processi, in quanto si è scelto di usare un unico segmento per i processi, che occupa buona parte di tutta la memoria virtuale disponibile (per essere precisi, in realtà i segmenti sono due, uno per il codice e uno per i dati, sovrapposti).

I "segmenti logici" dei programmi (come codice, dati, stack, librerie condivise ecc.) sono invece gestiti via software e i relativi diritti di accesso tramite la protezione offerta a livello di pagine. Questo ha il vantaggio di consentire una portabilità maggiore su altre architetture. La **LDT** può comunque essere modificata tramite l'apposita system call `modify_ldt()` per

esigenze particolari (per esempio sull'architettura x86 viene utilizzata dagli emulatori di altri "sistemi operativi", quali il DOS con il DOSEMU, il Windows con Wine e Wabi).

Il TSS è una struttura dati manipolata direttamente dal processore, che questo utilizza per salvare lo stato del processo e per ripristinarlo durante i context switching: è dunque necessario riservarne una diversa per ogni processo.

3.3 Livelli di privilegio

E' noto che tutti i sistemi operativi Unix prevedono almeno due livelli distinti di protezione: uno per il kernel ed uno per il funzionamento vero e proprio dei processi. Linux utilizza per la protezione della memoria due dei quattro livelli di privilegio offerti dai processori x86 a livello di segmentazione: il modo kernel corrisponde al funzionamento con Current Privilege Level pari a 0 (che è il massimo privilegio possibile), mentre il modo user a CPL pari a 3 (il minimo). Questo significa che certe operazioni critiche per la sicurezza del sistema potranno essere eseguite solamente quando i processi si trovano in modo kernel, come d'altronde è logico che sia.

Inoltre in questo modo tutti i tentativi intenzionali o non di utilizzare istruzioni pericolose per il sistema da parte dei processi scatenano un'eccezione tramite la quale il kernel riprende il controllo e applica gli opportuni provvedimenti.

3.4 Gli interrupt

La gestione delle interruzioni è un meccanismo fondamentale di qualsiasi sistema a microprocessore. In particolare tramite esse è possibile ottenere un'efficiente gestione dei dispositivi periferici da parte del sistema operativo. Nei processori della famiglia x86 funzionanti in modo protetto a tal fine è indispensabile preparare una struttura dati chiamata Interrupt Descriptor Table (IDT) contenente 256 gate corrispondenti ai 256 interrupt possibili. Nella figura 3.3 si vede uno schema riassuntivo dei gate utilizzati e la loro funzione.

Si può notare come nella **IDT** siano presenti due tipi di gate: l'interrupt gate ed il trap gate. Il primo tipo è utilizzato per la gestione degli interrupt request dei dispositivi esterni, mentre il secondo è utilizzato in tutti gli altri casi, ossia: la gestione delle trap definite o riservate dalla Intel (dalla 0 alla 31); la gestione delle system call (interrupt 80 hex); la gestione degli eventuali interrupt non definiti.

Per quanto riguarda le traps e le system call, i gate corrispondenti sono impostati nel file arch/i386/kernel/traps.c, mentre per gli IRQ bisogna riferirsi alle routine di inizializzazione dei vari dispositivi periferici.

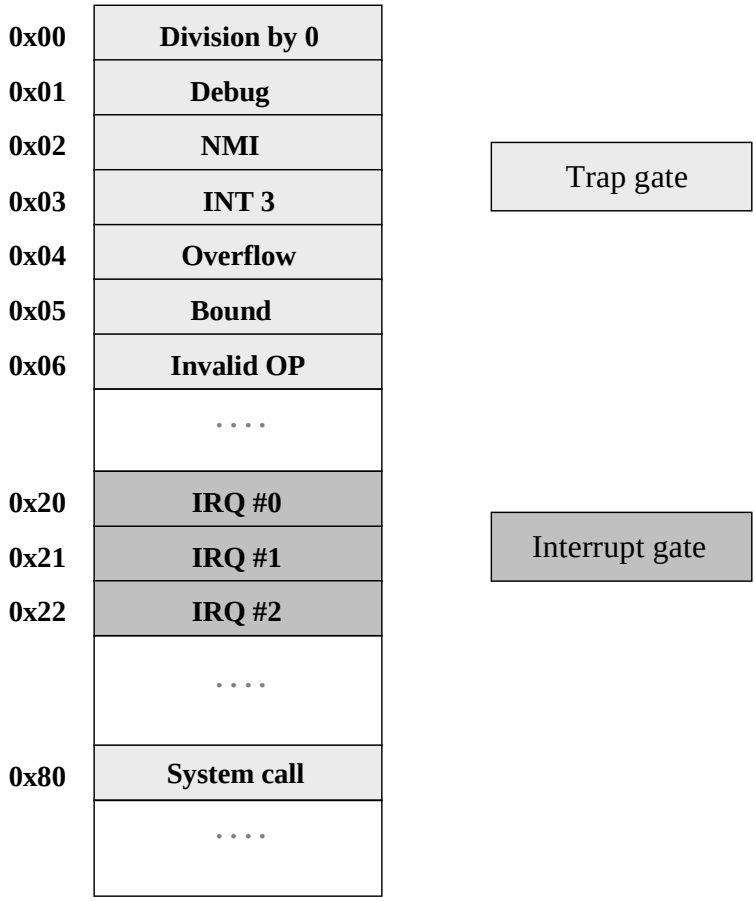

Figura 3.3: Gli interrupt

Si noti che Linux non utilizza in alcun modo i task gate, previsti dall'architettura x86 per consentire un context switching automatico al verificarsi dell'interrupt corrispondente.

3.5 System calls

Come avviene generalmente in tutti gli UNIX, le system call sono mappate tutte su un unico interrupt (il numero 80 in esadecimale su Linux), e vengono selezionate tramite un numero unico per ogni funzione. In particolare il numero della funzione richiesta viene posto nel registro EAX, ed i parametri sono posti negli altri registri, in particolare in EBX, ECX, EDX, ESI, EDI. Sarebbe più scomodo metterli sullo stack in quanto dopo l'esecuzione dell'istruzione int lo stack cambia e diventa quello del livello 0, non direttamente accessibile dal chiamante. Comunque la routine agganciata all'`INT 0x80` provvede per prima cosa a

salvare i registri sullo stack, cosicch le funzioni C se li ritrovino come parametri.

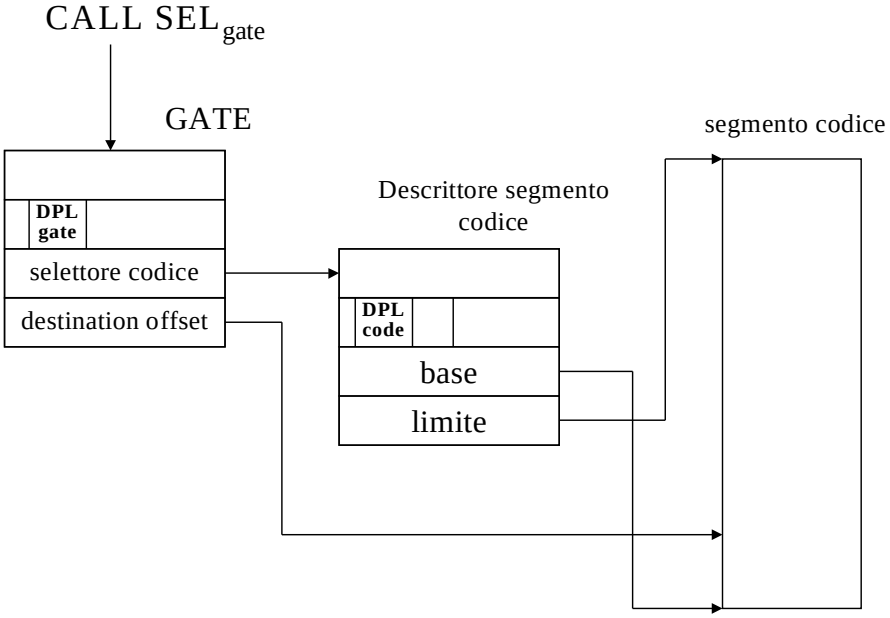

Figura 3.4: Utilizzo dei GATE

Per poter provocare il cambio del livello di privilegio, è necessario avvalersi del gate posizionato nella **IDT**, che ha ovviamente DPL pari a 3 per poter essere acceduto da qualsiasi programma. Il gate inoltre forza sia il selettore che l'offset all'interno del segmento codice di destinazione ad un valore ben preciso, ossia il punto di ingresso della procedura assembler di gestione delle system call che si occupa di capire quale funzione è stata richiesta e di richiamare la corrispondente routine in linguaggio C. Per maggiori particolari si può vedere il file arch/i386/kernel/entry.S, in cui si trova il codice.

Si ricorda che il gate è la "porta" predisposta dai processori della famiglia x86 tramite la quale è necessario passare per effettuare chiamate interlivello; essa inoltre consente al microprocessore di fare tutte le verifiche di accessibilità del caso e di predisporre i vari registri critici con un valore opportuno, tale da non mettere a rischio la sicurezza del sistema, come si evince dalla figura 3.4: In questo caso devono valere le relazioni:

accesso al gate:

$$Max(CPLatt, RPLgate) <= DPLgate$$

accesso al codice:

$$DPLdesc <= CPLatt$$

Si noti che per certe system call necessarie per esempio nel caso di emulazione di altri

sistemi, è stato necessario predisporre anche un call gate, posizionato nella **LDT**.

Poich lo stack non è utilizzato, la possibilità di copia dei dati dallo stack di livello inferiore a quello superiore prevista dai processori x86 non è utilizzata, ed il campo corrispondente (dword counter) nel gate è posto a zero.

Capitolo 4

La paginazione

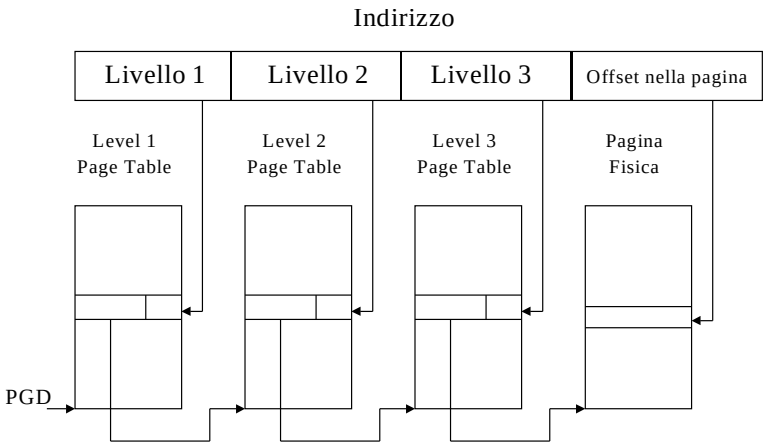

Figura 4.1: Le page table

In Linux l'organizzazione della paginazione è stata virtualizzata, ossia il sistema implementa uno schema generale che viene di volta in volta adattato alle varie architetture. Questo prevede tre livelli di page table: ognuna di esse contiene il numero della pagina corrispondente alla page table del livello successivo. Per tradurre l'indirizzo virtuale in un indirizzo fisico, il processore deve dunque prendere il contenuto di ciascun campo in cui l'indirizzo virtuale è stato suddiviso, convertirlo in un'offset all'interno della pagina fisica che contiene la page table e leggere l'indirizzo della pagina corrispondente alla page table del livello successivo. Questa operazione è ripetuta tre volte, finch si trova l'indirizzo della pagina fisica cui corrisponde l'indirizzo virtuale dato; si utilizza infine l'offset iniziale per accedere all'interno della pagina alla parola di dato richiesta. Nella figura 4.1 si vede uno schema di quanto appena descritto.

Ogni piattaforma su cui Linux è stato portato fornisce le sue particolari macro di traduzione per attraversare le page table: in questo modo il kernel non necessita di conoscere il formato delle stesse o come queste sono organizzate. Il meccanismo funziona talmente bene che Linux può usare lo stesso identico codice di manipolazione delle pagine per tutte le architetture, siano esse per esempio l'Intel x86 che ha due livelli di paginazione oppure l'Alpha o lo Sparc, che invece hanno tre livelli.

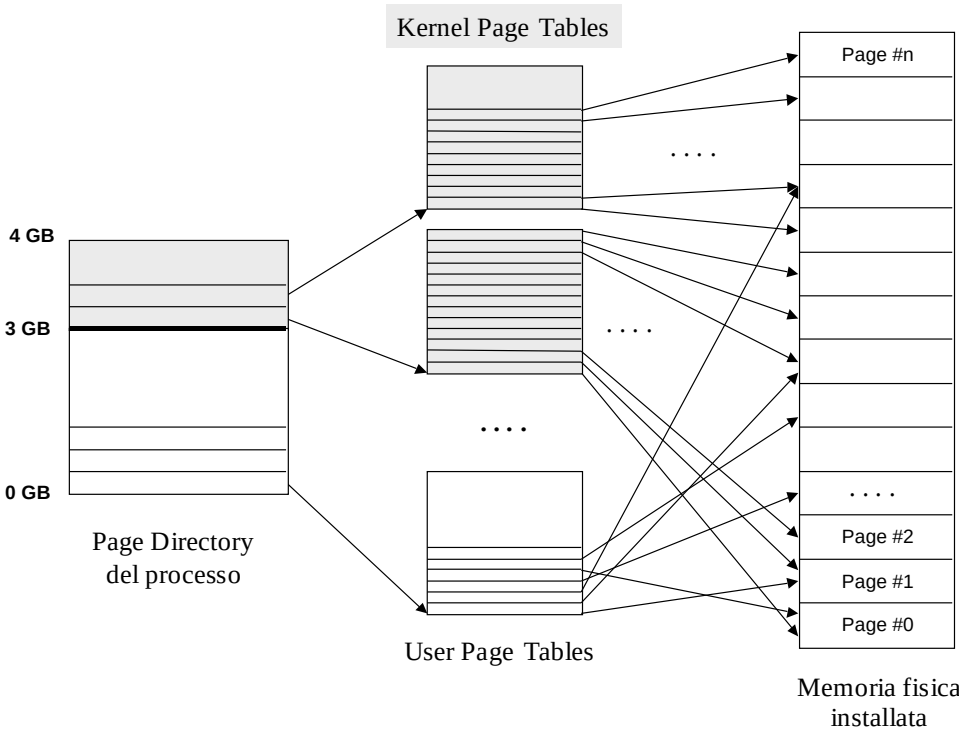

Figura 4.2: Kernel Page Tables

In Linux si è scelto di assegnare ad ogni processo una sua page directory propria, il cui indirizzo viene caricato nell'apposito registro del processore ad ogni cambio di contesto. La parte utilizzata dai processi quando questi operano in kernel mode contiene comunque gli stessi valori per tutti, come è d'altronde logico aspettarsi. Qui di seguito vediamo in dettaglio quanto appena descritto. La page directory (ossia il primo livello di page table) di ogni processo, come si vede dalla figura 4.2, è logicamente suddivisa in due parti.

La prima parte, corrispondente agli indirizzi lineari da 0 GB a 3 GB, contiene le informazioni di traduzione per le pagine utilizzate dal processo quando si trova in "user mode". La seconda parte, che comprende gli indirizzi lineari da 3 GB a 4 GB, contiene le informazioni usate per la traduzione in "kernel mode" (gli indirizzi corrispondono infatti ai segmenti del kernel). Si può notare la corrispondenza ordinata tra le page table entry e le

pagine nella memoria fisica. Le informazioni memorizzate nelle page table hanno infatti la particolarità di far corrispondere ad un certo indirizzo virtuale del segmento del kernel il medesimo indirizzo fisico in memoria. Questa corrispondenza è stabilita, per i primi 4 MB di memoria, nel file arch/i386/kernel/head.S in fase di inizializzazione, e completata poi nel file arch/i386/mm/init.c per tutte le pagine necessarie a coprire completamente la memoria fisica installata sul particolare sistema su cui il kernel si sta caricando.

Questa scelta è causata dal fatto che è il kernel che gestisce le pagine dei processi in modo user, e se esso dovesse gestire anche le pagine per se stesso, questo comporterebbe una notevole complicazione, se non addirittura l'impossibilità di adempiere al compito.

4.1 Page allocation

In un sistema le pagine fisiche in memoria devono adempiere a svariati compiti: per esempio poter essere allocate e deallocate, contenere le strutture dati del kernel, e le page table stesse. Il meccanismo di allocazione e deallocazione è fondamentale, in quanto risulta critico per l'efficienza del sottosistema di gestione della memoria virtuale.

L'organizzazione di Linux prevede che tutte le pagine fisiche siano descritte da una struttura dati di nome `mem_map` (definita in mm/memory.c), che è un vettore di strutture `mem_map_t` inizializzata durante il caricamento del sistema, nel file mm/page_alloc.c, funzione `free_area_init()`. Ogni `mem_map_t` descrive una singola pagina fisica. I campi più importanti sono: count numero degli utenti della pagina: se è maggiore di uno, allora la pagina è condivisa tra più processi; age riporta l'età associata alla pagina, ed è usato per decidere se questa è una buona candidata per essere eliminata o portata nello swap; `map_nr` il numero della pagina fisica che questa struttura descrive.

La struttura è definita nel file include/linux/mm.h. Esiste inoltre un vettore `free_area`, utilizzato dalla routine di allocazione delle pagine per trovare quali siano libere. Ogni elemento di questo vettore contiene una mappa di bit tramite la quale si determinano quali siano i blocchi di pagine liberi, oltre che la testa delle liste che descrivono i blocchi da una pagina, i blocchi di due pagine, i blocchi di quattro e cosìvia, in potenze di due. Tutte le pagine libere vengono accodate in liste puntate dal vettore `xfree_area`. Di seguito si illustrano graficamente le strutture dati descritte.

Linux usa un algoritmo particolare per allocare e deallocare le pagine, detto Buddy Algorithm. Innanzi tutto la routine alloca solamente blocchi di pagine (di 1, 2, 4, 8 pagine, come descritto in precedenza): essa per prima cosa cerca i blocchi di dimensione corrispondente alla richiesta, e se non ne trova, cerca quelli di dimensione superiore (ossia il doppio), finch tutta la `free_area` è stata passata in rassegna o un blocco di pagine è stato trovato. Se il blocco è più largo della richiesta, esso viene suddiviso finch diventa della dimensione

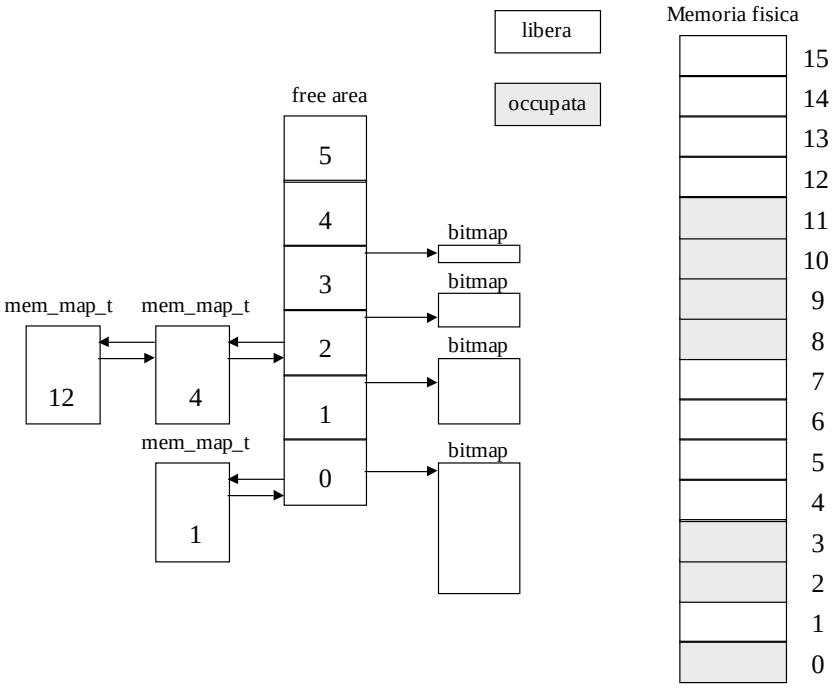

Figura 4.3: Gestione della memoria

giusta. Grazie alle dimensioni dei blocchi una il doppio dell'altra, quest'operazione è semplice, infatti basta continuare a suddividere i blocchi a metà. I blocchi liberi cosìgenerati sono aggiunti alla coda appropriata e il blocco allocato è ritornato al chiamante.

La deallocazione tende a frammentare la memoria in piccoli blocchi, ma la routine cerca di ricombinare le pagine in blocchi più grossi quando ciò è possibile. Anche in questo caso le particolari dimensioni scelte facilitano notevolmente l'operazione. Quando un blocco viene deallocato il codice verifica per prima cosa se quello adiacente è libero: in tal caso li unisce, e questo procedimento si ripete ricorsivamente, fino a che ciò è possibile; cosìfacendo, i blocchi liberi sono i più grossi che la memoria consenta di avere. Per maggiori dettagli si può visionare il file mm/page_alloc.c, ed in particolare la funzione `free_pages_ok()`.

4.2 L'organizzazione della memoria di un processo

Ad ogni processo nel sistema Linux associa una struttura dati di nome `task_struct` (vedi include/linux/sched.h), contenente a sua volta un campo mm che punta a una `mm_struct` (vedi include/linux/sched.h), la quale contiene dei puntatori ad un certo numero di strutture di tipo `vm_area_struct` (file include/linux/mm.h).

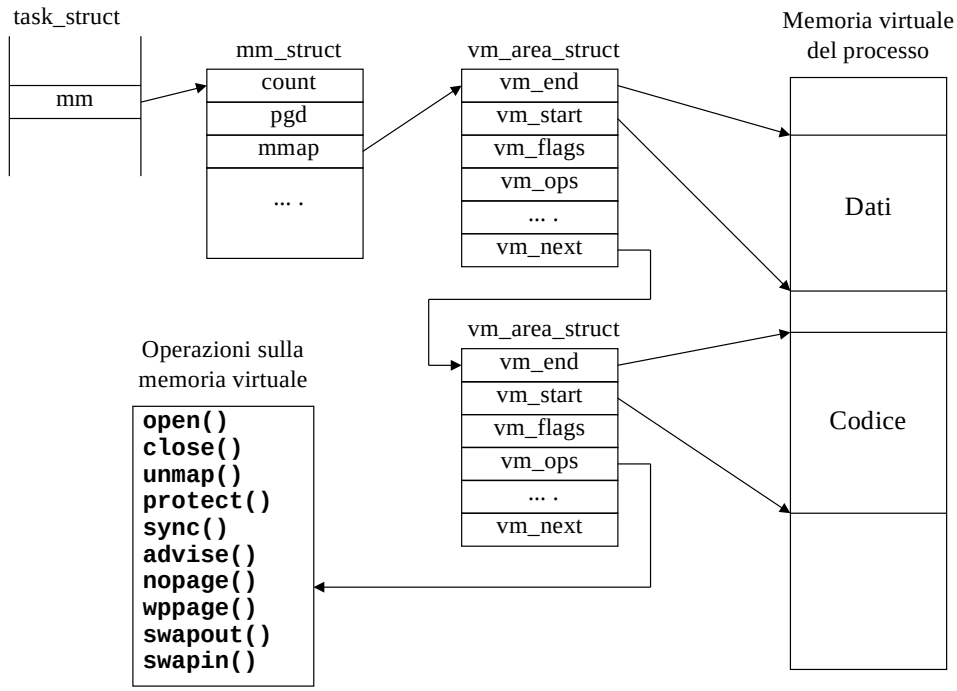

Figura 4.4: Memoria di un processo

Ogni `vm_area_struct` descrive un'area di memoria virtuale destinata a qualcosa, e contiene tra le altre cose l'indirizzo di inizio e fine, i permessi di accesso da parte del processo e un insieme di operazioni eseguibili sulla memoria, contenute in una `vm_operations_struct` (file include/linux/mm.h), che, se definite, Linux utilizza quando deve operare su quella particolare porzione di memoria.

Per avere un esempio di un possibile impiego di dette operazioni, si pensi ad un processo che accede a una zona di memoria la cui corrispondente pagina fisica non è presente: questo comporta un'eccezione di page fault; la routine riconosce la situazione e decide di usare l'operazione associata di nome nopage.

Nel caso l'area di memoria si riferisca ad un'immagine eseguibile, nopage sarà stata precedentemente inizializzata a puntare ad un'apposita routine in grado di caricare la corrispondente pagina dal file.

Quando un eseguibile viene mappato in memoria per essere successivamente eseguito, viene generato un insieme di `vm_area_struct`, ognuna delle quali rappresenta una parte dell'immagine eseguibile: il codice vero e proprio, le variabili inizializzate, quelle non inizializzate e cosìvia.

Linux supporta un certo numero di operazioni standard sulla memoria virtuale, come si

può vedere dalla figura 4.4, e al momento della creazione di ogni `vm_area_struct`, associa loro l'insieme corretto e specifico di operazioni.

4.3 Il page fault

La routine del page fault (corrispondente alla trap n. 14 dei sistemi x86), è il principale strumento di gestione della memoria. Infatti Linux fa un uso estensivo dell'eccezione del page fault per assolvere a svariati compiti, e questa organizzazione, come si vedrà, permette anche una notevole efficienza. Il codice sorgente della routine si trova in arch/i386/mm/fault.c.

Essa viene richiamata tramite un'eccezione scatenata dal processore quando questo rileva tramite le tabelle di paginazione che la pagina a cui si cerca di accedere non è presente in memoria, oppure l'accesso non è conforme con i permessi previsti.

In tutti i sistemi la routine di page fault per prima cosa ricava in qualche modo l'indirizzo virtuale che ne ha causato l'esecuzione, la presenza o meno della pagina e il tipo di accesso tentato (lettura o scrittura e a quale livello di privilegio). Nei sistemi x86 in particolare, la prima informazione si ricava dal registro CR2 del processore, mentre la seconda da un apposito codice di errore posto sullo stack.

Per poter effettuare una corretta gestione del page fault è necessario che Linux per prima cosa individui la `vm_area_struct` che rappresenta l'area di memoria di cui l'indirizzo virtuale fa parte. Poich la velocità di ricerca in queste strutture dati è critica per ottenere una buona efficienza a tempo di esecuzione, queste sono organizzate in una struttura ad albero particolare, detta AVL tree (Adelson-Velskii e Landis). Essa è in pratica un albero binario, in cui sono mantenute alcune proprietà, e che permette di effettuare la ricerca di una chiave nell'albero in un tempo O(log n) al posto del comune O(n) di una normale lista linkata, dove n è ovviamente il numero di aree di memoria virtuale del task (generalmente 6 ma in alcuni casi si possono raggiungere le 3000). Questa caratteristica è dovuta alle proprietà dell'albero, che sono qui brevemente riassunte. Ogni nodo ha tre campi fondamentali: puntatore al figlio sinistro, puntatore a quello destro, e altezza. Nell'albero valgono sempre queste regole:

- l'altezza di un nodo è pari al massimo di quella dei suoi due nodi figli aumentata di una unità;
- la differenza tra l'altezza del figlio sinistro e quella del figlio destro non eccede l'unità;
- per ogni nodo nell'albero del figlio di sinistra, le chiavi contenute sono minori o uguali di quelle del padre e analogamente per il figlio di destra.

Queste proprietà sono mantenute richiamando la funzione di ribilanciamento dell'albero (`avl_rebalance`) ogni volta che questo viene modificato. Per ulteriori dettagli, si consiglia di vedere il file mm/mmap.c.

Una volta che è stata effettuata la ricerca in questo albero, si può presentare il caso che non ci sia nessuna struttura dati corrispondente all'indirizzo virtuale: questo significa che il processo ha tentato di accedere ad un indirizzo illegale; il kernel invia dunque al task un segnale di SIGSEGV (Segmentation fault), e se il processo non ha definito un handler per questo segnale, esso viene terminato. Se invece l'indirizzo è valido, occorre verificare se il tipo di accesso è compatibile con quelli permessi per quell'area di memoria. Ne caso in cui non sia così, si ricade nella condizione precedente, e si segnala al processo un errore di memoria.

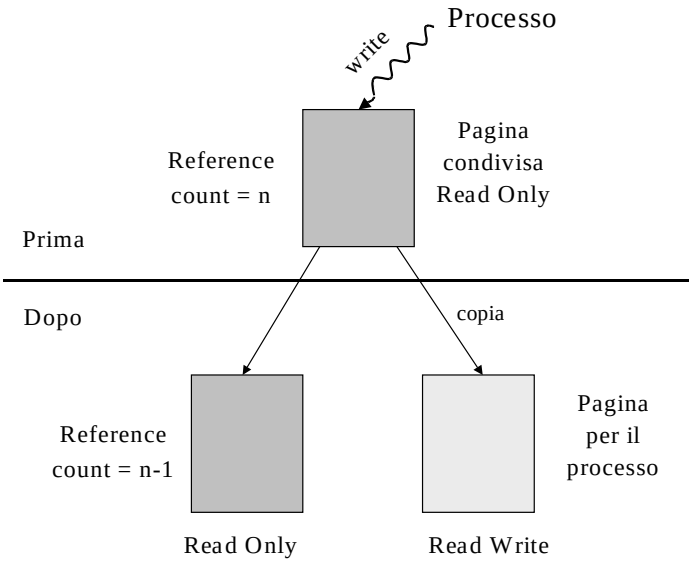

Figura 4.5: Copy on write

Se tutti i controlli hanno dato esito positivo, Linux deve ora servire il page fault. Per prima cosa determina se la pagina è in memoria fisica o no. Per far ciò si testa il bit di presenza, determinando se la pagina manca perch è stata portata nello swap o solamente perch non è mai stata creata. Nel primo caso, se è stata associata un'operazione specifica al puntatore swapin nella `vm_operations_struct` della `vm_area_struct` la si esegue, altrimenti si sua la funzione `swap_in()` di default (si veda la `do_swap_page()` nel file mm/memory.c).

Nel secondo caso, se tramite il puntatore nopage della struttura `vm_ops` è stata specificata una routine particolare per quest'area di memoria, la si richiama, altrimenti si usa una procedura di default (si veda la funzione `do_no_page()` in mm/memory.c).

Si noti che in entrambi i casi si cerca di condividere la pagina con una esistente se possibile, al fine di allocare una nuova pagina solo se strettamente necessario. Inoltre, se la pagina è condivisa, viene posta comunque read-only.

Ora che la pagina è presente in memoria, si aggiorna la rispettiva entry nella page table cosicch il processo possa riprendere la sua naturale esecuzione. Se invece la pagina è presente in memoria fisica ed è stato tentato un accesso in scrittura, significa che essa era protetta dalla scrittura. Poich i controlli sui permessi sono già stati eseguiti, vuol dire che la pagina è scrivibile ma condivisa per evitare speco di risorse. Di conseguenza se la pagina ha un reference counter maggiore di uno, si ricopia questa in una nuova, si decrementa il contatore della vecchia e si assegna la nuova pagina al processo che ne ha richiesto la scrittura. La situazione è schematicamente illustrata nella figura 4.5.

Quest'ultimo meccanismo è noto come "copy on write" e permette un notevole risparmio di pagine, soprattutto quando vengono lanciate più copie dello stesso eseguibile. In entrambi i casi è ovviamente necessario modificare la page table, perciò devono essere anche aggiornate le loro copie nelle varie cache del sistema (per esempio nel TLB).

4.4 On–demand–loading

Il meccanismo di *on–demand–loading* è molto utile per avere un efficiente esecuzione dei programmi. Quando si richiede l'esecuzione di un'immagine (ossia si lancia un programma), il contenuto della immagine eseguibile deve essere portato nello spazio di indirizzamento virtuale dei processi. Analoga cosa accade nel caso delle librerie condivise che devono essere utilizzate. Il file eseguibile però non viene copiato subito nella memoria fisica, bensìè semplicemente aggiunto nello spazio della memoria virtuale del processo: vengono cioè create delle `vm_area_struct` (la cui organizzazione è già stata vista in precedenza), che ricalcano la suddivisione del programma in codice, dati inizializzati, variabili non inizializzate, stack e cosìvia. Questa aggiunta delle varie parti del file allo spazio di indirizzamento del processo è conosciuta come operazione di "memory mapping".

Non appena il task viene poi selezionato dallo scheduler ed entra in esecuzione, tenta di accedere alla prima pagina della sua immagine eseguibile. Poich questa non è presente in memoria, l'hardware invoca la routine del page fault che provvede a caricare la pagina dal file del disco, con la sequenza di operazioni vista prima. L'esecuzione quindi può procedere finch si incontrerà un'altra pagina non presente. Questo meccanismo è particolarmente efficiente in quanto carica in memoria solamente le parti di codice che sono effettivamente eseguite dal programma, consentendo un certo risparmio di memoria.

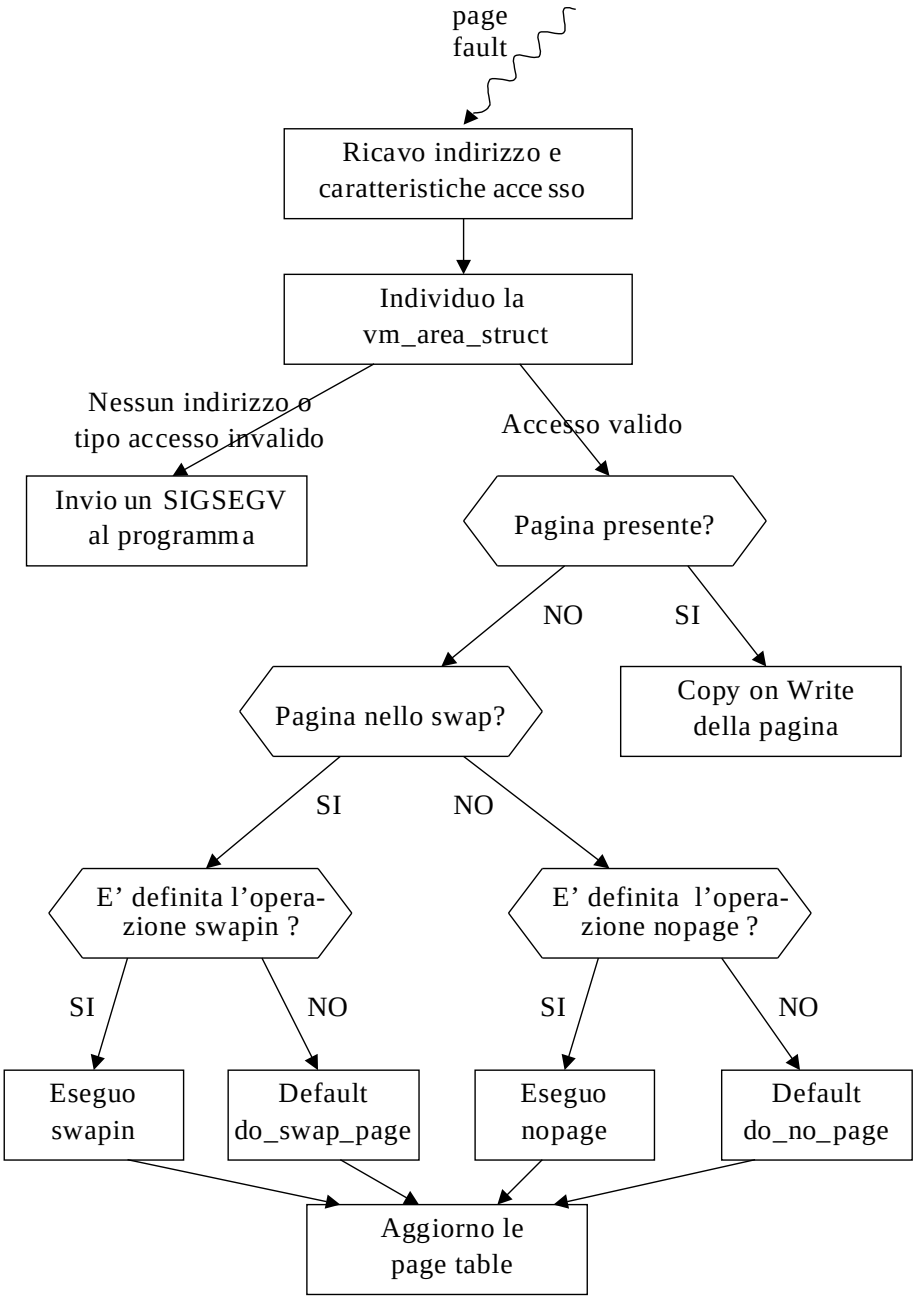

Figura 4.6: On–demand–loading

4.5 Page cache

Al fine di ottimizzare le prestazioni del sistema, Linux mantiene una page cache che contiene quelle utilizzate più recentemente. Questo meccanismo rende più efficiente l'*on demand*

loading appena visto nel caso delle immagini eseguibili.

Quando una pagina deve essere letta da un file mappato in memoria per prima cosa si interroga la page cache. Se in essa è presente la pagina cercata, è sufficiente restituire al page fault handler semplicemente il puntatore alla pagina; in caso contrario, essa verrà letta dal disco, assieme ad alcune pagine successive, tramite un'operazione di read ahead, ponendo le pagine non richieste nella page cache, pronte per essere rese disponibili al processo se questo sta per esempio accedendo ad esse in modo sequenziale. Col tempo, eseguendo sempre nuovi programmi, la page cache crescerà; quando non vi sarà più posto per le nuove pagine, quelle più vecchie e meno accedute saranno rimosse.

Questo meccanismo permette ai programmi di essere lanciati e ed avere prestazioni discrete anche se il numero di pagine fisiche disponibili è basso. Se necessario Linux è comunque in grado di diminuire le dimensioni di questa page cache.

4.6 Swapping

Come in tutti i sistemi di memoria virtuale, quando la memoria fisica disponibile diventa scarsa, Linux deve cercare di liberare delle pagine fisiche. Questa operazione viene affidata ad un apposito demone del kernel, chiamato kswapd. Esso è uno speciale tipo di processo, cioè un thread del kernel, che non ha memoria virtuale e gira in kernel mode: in questo modo può dunque manipolare direttamente la memoria fisica. Inoltre sempre a questo demone è affidato il compito di assicurarsi che ci siano abbastanza pagine libere in memoria per mantenere il sistema efficiente.

Il kswapd viene attivato dal processo del kernel stesso durante il caricamento, e generalmente è in stato dormiente, in attesa di essere risvegliato periodicamente dal kernel swap timer. Ogni volta che questo timer scade, il demone verifica se il numero delle pagine libere sta diventando troppo esiguo. A tal fine si avvale di due variabili, `free_pages_high` e `free_pages_low`, per decidere se liberare delle pagine. Fintantoch il numero di queste è maggiore di `free_pages_high`, non fa niente. Se invece il numero di pagine è inferiore a `free_pages_high` o peggio a `free_pages_low`, il thread cerca di diminuire le pagine usate dal sistema per esempio riducendo la dimensione del buffer cache e della page cache oppure portando nello swap alcune pagine di memoria. Le due variabili menzionate precedentemente servono per dare un'indicazione al kswapd su quante pagine alla volta deve cercare di liberare prima della successiva riattivazione. In genere il thread continua ad utilizzare lo stesso metodo (riduzione delle cache o swap delle pagine) usato la volta precedente se si è rivelato una buona scelta. Inoltre, se rileva che il numero di pagine è inferiore a `free_pages_low`, dimezza il periodo del timer che lo deve risvegliare.

Il buffer cache e la page cache sono in genere buone riserve a cui attingere pagine da

liberare: spesso infatti ne contengono di non strettamente necessarie, che consumano solo la memoria del sistema. Mediamente eliminare pagine da queste cache non richiede scritture sui device fisici (al contrario dell'operazione di swap), ed inoltre ciò non ha effetti troppo dannosi sul sistema se non quello di rendere gli accessi ai device fisici e ai file mappati in memoria un po' più lenti. Se comunque lo scarto delle pagine è fatto con una politica equa, tutti i processi subiranno lo stesso effetto. Per mettere in atto l'operazione si esaminano i blocchi di pagine in maniera ciclica, con il noto algoritmo dell'orologio, che permette di passare in rassegna tutte le pagine prima di ritornare su una precedentemente visitata.

Si noti che poich nessuna delle pagine che vengono liberate faceva parte della memoria virtuale di alcun processo (esse erano al massimo cached pages), nessuna tabella di paginazione necessita di aggiornamenti.

Diverso è il caso delle pagine a cui i processi accedono tramite la memoria virtuale. Se infatti una di queste viene spostata dalla memoria nel device di swap, la corrispondente tabella di paginazione deve essere aggiornata, ponendo a falso il bit di presenza, e indicando nei restanti bit il device di swap e l'offset nello stesso tramite il quale la pagina potrà essere in seguito recuperata quando questa necessiterà di essere referenziata.

Per decidere quali pagine portare nello swap, un questo caso Linux usa un meccanismo di aging: ogni pagina ha un contatore (che fa parte della struttura `mem_map_t`) che permette al demone di swap di capire se la pagina è una buona candidata o no per l'operazione. Questo contatore viene incrementato di tre unità fino ad un massimo di 20 quando si accede alla pagina, e decrementato di una ad ogni attivazione del demone. In questo modo si seleziona dunque una pagina vecchia, e se la pagina è "dirty", ossia è stata acceduta in scrittura, è necessario copiare effettivamente i suoi dati sul device di swap.

Se la pagina si trova in una regione di memoria virtuale per cui è stata definita una operazione particolare per operare lo swap (funzione swapout nella struct `vm_operations_struct` contenuta nella `vm_area_struct`), essa viene richiamata, altrimenti il demone di default alloca una pagina nello swap e vi copia la vecchia, aggiornando di conseguenza la page table interessata. La pagina vecchia viene poi resa libera accodandola nella struttura `free_area`.

Se la pagine non è "dirty", si può invece liberare senza ulteriori copie, in quanto ne esiste sicuramente già una salvata da qualche parte. Ovviamente questo metodo è molto più efficiente che portare nello swap interi processi, in quanto cerca di mantenere un equilibrio nel sistema con il minimo dispendio di risorse.

Si noti infine che invece le pagine contenenti il codice binario dei programmi eseguibili, essendo per loro natura a sola lettura, non vengono mai spostate sul device di swap, bensìsempre ricaricate dall'immagine su disco o dal buffer cache quando ciò è richiesto. Questo consente di occupare lo spazio destinato allo swap solamente con i dati delle applicazioni, ottenendo cosìun'efficienza notevole.

Un'ultima considerazione riguarda le entry delle page table corrispondenti alle pagine che si trovano anche nel device di swap: esse costituiscono una sorta di "swap cache", contenendo la posizione (device e offset) che le pagine occupano nello swap. Linux, se si accorge che la pagina non è stata modificata, consulta questa entry: se essa è vuota, significa che indispensabile scrivere la pagina sul device, ma se contiene un valore, vuol dire che la pagina è già sul device ed è dunque inutile riscriverla uguale: si risparmia in questo modo prezioso tempo.

4.7 Gestione del bug del pentium

Quando un processore della famiglia x86 incontra un'istruzione invalida, normalmente genera un'eccezione di "invalid opcode", ed il sistema operativo termina il programma. Se però questo meccanismo fallisce, il programma errato può bloccare l'intero sistema. Questo è ciò che accade quando il Pentium incontra il noto bug "F00F", che è causato dall'istruzione `LOCK CMPXCHG8B EAX`.

La versione normale di questa confronta una zona di memoria di 64 bit con il valore dei registri EAX ed EDX: uno degli operandi è dunque in memoria e l'altro è implicitamente in EDX:EAX. E' però possibile costruire una codifica dell'istruzione che non usa l'operando in memoria, bensìun registro. Poich l'istruzione è invalida, essa non viene mai generata dai compilatori, però è possibile costruirla manualmente. Finch non si antepone il prefisso `LOCK`, tutto funziona ed il processore genera l'eccezione correttamente; con la `LOCK` invece, il Pentium si blocca compromettendo cosìl'intero sistema. L'unico modo per recuperare da questa situazione di stallo è il reset.

Il fatto è particolarmente grave in quanto l'istruzione non è privilegiata, quindi può essere eseguita anche in modo user da qualsiasi utente, pregiudicando cosìla stabilità dell'intero sistema. Dopo circa una settimana dalla scoperta del bug, fortunatamente l'Intel ha annunciato di aver messo a punto una possibile soluzione che poteva essere incorporata praticamente in qualsiasi sistema operativo (eccetto quelli in modo reale come il DOS). Col tempo si sono poi scoperti altri metodi non ufficiali che risolvono il problema con minori sforzi.

La sequenza di operazioni che porta al blocco è pressappoco la seguente: quando il processore incontra l'istruzione `F00FC7C8`, esso riconosce che è un'istruzione invalida e cerca di richiamare l'handler dell'"invalid opcode". Il prefisso `LOCK` però confonde il processore: questo infatti asserisce erroneamente il segnale `LOCK`, che però deve essere utilizzato solo nei cicli read-modify-write. Il processore invece effettua due letture consecutive per recuperare l'indirizzo dell'handler "invalid opcode", senza effettuare una scrittura in mezzo, per cui rimane in uno stato strano e si blocca da solo. Proprio grazie alla `LOCK` che fa credere al

processore in un primo tempo di dover effettuare la scrittura, si sono potute trovare delle soluzioni al problema.

Una operazione comune a tutti rimedi è quella di allineare in modo particolare la **IDT** sul limite di una pagina fisica di memoria, facendo in modo di avere le prime 7 entry della tabella in una pagina e tutte le altre nella successiva. Le varie soluzioni differiscono poi sugli attributi particolari da dare a questa pagina.

La prima soluzione Intel propone di segnare la pagina come non presente, in modo che ad ogni tentativo di accesso venga generato un page fault. Questo ha però lo svantaggio che tutte le volte che si genera un'eccezione tra le prime 7 dell'architettura x86, venga eseguito sempre un page fault, che deve preoccuparsi di controllare i livelli di privilegio, fare tutti gli altri controlli del caso e verificare se la routine è stata invocata al posto di un'atra eccezione, nel qual caso deve richiamarla, risistemando lo stack in maniera opportuna. Dal punto di vista del sistema operativo, questa non è certo la miglior soluzione.

La seconda invece prevede, oltre al particolare allineamento, di marcare la pagina con le prime 7 entry a sola lettura (e ovviamente di settare il flag WP del processore, per intercettare le scritture su pagine read-only anche in supervisor mode). In questo caso la routine del page fault deve solamente verificare l'indirizzo che ne ha causato l'esecuzione (contenuto nel registro CR2), e verificare se è quello corrispondente all'eccezione di "invalid opcode". Se è così, dopo aver sistemato lo stack, richiama l'handler relativo, altrimenti procede con le consuete operazioni. Questa soluzione ha il vantaggio che se si verifica una delle prime sei eccezioni, la routine del page fault non viene interessata in alcun modo. Il bug fa funzionare questo metodo perch tramite la `LOCK` il processore pensa erroneamente di dover modificare la pagina di memoria della **IDT** con il gate corrispondente all'"invalid opcode", e di conseguenza viene attivato il page fault handler.

Le soluzioni non ufficiali scoperte col tempo hanno rivelato che nel meccanismo del bug è coinvolta la cache di primo livello del processore. Infatti, rendendo la solita pagina non cacheable (settando il PCD bit (page cache disable) nella page table entry corrispondente) oppure costringendo il processore ad adoperare un meccanismo di Write Through per quella pagina (settando il PWT bit (page write through) sempre nella page table), si ottiene che il bug magicamente sparisce, senza ulteriori modifiche alle varie routine del sistema operativo. Questo metodo consente di trovare una soluzione anche per "sistemi operativi" in modo reale quali il DOS: è sufficiente disabilitare la cache del processore dal BIOS del computer. Questo è ovviamente piuttosto penalizzante per le prestazioni, ma si presume comunque che utilizzando il DOS su un processore Pentium non si pretendano prestazioni eccezionali.

Linux utilizza il metodo di proteggere la pagina della **IDT** con il bit di sola lettura. Questo viene impostato durante l'inizializzazione del sistema: per prima cosa viene richiamata la routine `check_bugs()` nel file init/main.c, che provvede ad effettuare numerosi controlli

sul funzionamento del sistema in genere, tra cui dei test sul coprocessore e sull'istruzione hlt, e che infine chiama la `check_pentium_f00f()`, la quale verifica se è presente il difetto, assumendo che lo sia sempre se la CPU è un processore Intel originale.

Se il bug è presente, viene chiamata `trap_init_f00f_bug()` in arch/i386/kernel/traps.c che si preoccupa di rendere la pagina della **IDT** a sola lettura. L'ulteriore modifica del sistema operativo necessaria si trova nella routine del page fault (arch/i386/kernel/mm/fault.c): essa tra le varie operazioni, se la pagina acceduta è read-only, verifica se l'indirizzo che ne ha scatenato l'esecuzione è quello corrisponde alla settima entry nella **IDT**, nel qual caso richiama la routine di gestione dell'`invalid opcode` (do_invalid_op()). Con queste poche modifiche si è reso dunque Linux immune a questo pericoloso bug dei processori Intel. Infine si illustra la situazione della **IDT** dopo la protezione.

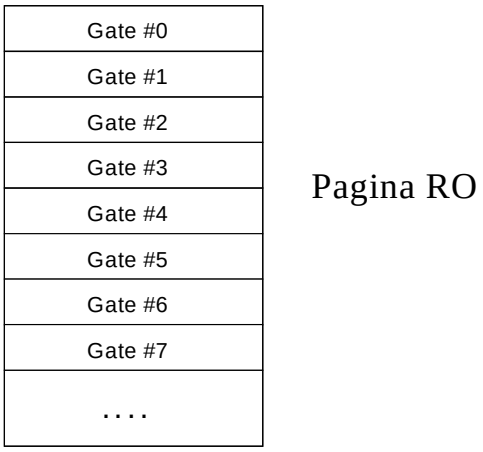

Figura 4.7: Protezione della IDT

Capitolo 5

Context switching

Uno dei meccanismi fondamentali in un sistema operativo è sicuramente quello del context switching, ossia l'operazione tramite la quale il processore salva il suo stato corrente (relativo al processo in esecuzione) e carica quello del prossimo processo (che è stato determinato in precedenza dallo scheduler). Affinch il context switching possa avvenire è indispensabile che il processore cambi livello di privilegio.

Tutti i processi in Linux girano parzialmente in user mode e parzialmente in kernel mode. Indipendentemente dall'architettura, generalmente c'è un meccanismo sicuro per passare dalla prima modalità alla seconda e viceversa. Ogni volta che un processo esegue una system call, entra in kernel mode ma continua ad essere eseguito: la differenza è che a questo punto il kernel sta girando per conto di quel processo. Ai processi non è consentito di interrompere il task corrente, ma invece ogni processo decide di rilasciare la CPU su cui sta girando quando deve attendere che si verifichi qualche evento da parte del sistema: per esempio, può mettersi in attesa di un byte da un file. Questa attesa avviene sempre all'interno di una system call, in kernel mode. In questi casi il processo viene sospeso ed un altro viene selezionato per essere eseguito. Nonostante i processi eseguano spesso delle system call, non è detto che lo facciano con regolarità, quindi Linux usa uno scheduling di tipo preemptive al fine di ottenere un buon multitasking. In quest'ottica, a ogni processo è permesso di girare per un certo time-slice; quando questo scade, viene posto in attesa per un po' finch non è nuovamente selezionato. Quest'ultima operazione è ovviamente compito dello scheduler, che in Linux usa il classico algoritmo Round Robin (si veda il file kernel/sched.c).

A questo proposito, il kernel di Linux prevede che, una volta assolti i compiti per cui è stato richiamato (per esempio per una system call o per servire un interrupt), prima di restituire il controllo al processo in user mode, verifichi se debba essere richiamato lo scheduler (tramite la variabile `need_resched`), che determina quale processo ha diritto di

adoperare la CPU nel prossimo time-slice. Se il processo schedulato è diverso da quello corrente viene invocata la routine che si preoccupa di eseguire materialmente il context switching (la macro `switch_to()` nel file include/asm/system.h), altrimenti si ritorna in user mode tramite l'istruzione iret (si veda la macro `RESTORE_ALL` in arch/i386/kernel/entry.S).

5.1 La routine

E' interessante analizzare brevemente il codice assembler utilizzato dalla routine. Essa, per quanto riguarda l'architettura x86, sfrutta appieno i meccanismi del processore per salvare e ripristinare lo stato dei processi. Nel seguito sono riportate le poche righe di codice che svolgono il compito.

```
__asm__("movl %2,"SYMBOL_NAME_STR(current_set)"nt" \
"ljmp %0nt" \
"cmpl %1,"SYMBOL_NAME_STR(last_task_used_math)"nt" \
"jne 1fnt" \
"cltsn" \
"1:" \
: /* no outputs */ \
:"m" (*(((char *)&next->tss.tr)-4)), \
 "r" (prev), "r" (next));
```

La ljmp è l'istruzione che consente il context switching. Nei processori Intel infatti ciò può essere ottenuto tramite una qualsiasi di queste quattro operazioni: jmp o call che si riferisce a un descrittore di TSS; jmp o call che si riferisce a un task gate; un interrupt o eccezione che punti a un task gate nella **IDT**; il task corrente esegue una iret con il flag NT settato.

La scelta dei progettisti di Linux è stata la prima, ossia il salto utilizzando come destinazione il descrittore del TSS del nuovo processo.

Non è di immediata comprensione il valore a cui la ljmp punta: per capirlo bisogna notare che la ljmp (corrispondente a una JMP FAR in assembler Intel) in modo protetto può ricevere come parametro un indirizzo di memoria di 32 bit, che contiene l'offset della nuova locazione su 4 byte e il descrittore del nuovo segmento su 2 byte. Ovviamente, poich il descrittore si riferisce al TSS del nuovo processo, l'offset viene ignorato, non avendo in questo caso alcun significato (il nuovo EIP viene caricato dal TSS). L'offset -4 specificato nella penultima riga è dovuto alla necessità di comprendere anche questo offset non utilizzato, che corrisponde a 4 byte di un campo fittizio aggiunto tra la fine dell'I/O map ed il campo

tr (infatti la `io_bitmap` è una doubleword più grande del dovuto). Per avere una conferma di ciò si può esaminare la struttura `thread_struct` nel file include/asm/processor.h.

Potrebbe stupire la presenza di codice di seguito all'istruzione di ljmp, ma si consideri che quando viene salvato lo stato del processo da interrompere, si salvano anche CS e EIP correnti, che puntano all'istruzione successiva alla ljmp: quindi, quando il processo verrà nuovamente caricato, continuerà la sua esecuzione dall'istruzione successiva alla ljmp, su cui era stato fermato precedentemente.

5.1.1 Problema del coprocessore

Si noti che subito dopo la ljmp è presente l'istruzione clts per azzerare il flag che segnala un context switching. Questo bit è utilizzato per i casi in cui avviene un context switching e successivamente il coprocessore segnala un errore: questo va però riportato non al processo corrente, ma a quello precedente. Di ciò si tiene traccia tramite questo bit, che viene impostato automaticamente dall'hardware senza ulteriori operazioni in caso di context switching.

Capitolo 6

Gestione dei processi

Al momento del caricamento del sistema l'unico processo è quello iniziale che gira ovviamente in kernel mode. Come tutti gli altri processi, anche questo ha uno stato, che è rappresentato dai registri, dallo stack e cosìvia, che sono salvati nella `task_struct` corrispondente quando non appena vengono creati e fatti girare altri processi. Alla fine dell'inizializzazione del sistema viene lanciato un kernel thread di nome init, che è composto da un ciclo infinito che non esegue nessuna istruzione utile: quando non c'è nient'altro da fare, lo scheduler seleziona questo processo. La `task_struct` di questo processo è l'unica che non viene allocata dinamicamente, ma che invece è determinata a tempo di compilazione nel kernel. Nei sorgenti ci si riferisce ad essa con il nome di `init_task`. Si noti che in questo processo, tramite l'istruzione di hlt, il kernel di Linux è in grado di mettere il processore in modalità di basso consumo energetico.

Il processo init ha PID (process identifier) pari a uno, ed è il primo processo del sistema. Esso esegue alcuni compiti iniziali, come aprire la console di sistema, montare il root file system, e lanciare il programma di inizializzazione del sistema, generalmente /sbin/init o simili. Inoltre usa il file /etc/inittab come script per creare nuovi processi, per esempio il getty che crea il login usato per autenticarsi sul sistema.

Per tener traccia dei vari processi presenti nel sistema, Linux definisce un vettore di puntatori a delle strutture `task_struct`, ognuna delle quali è creata in concomitanza al corrispondente processo (si veda il file include/linux/sched.h e kernel/sched.c). Il massimo numero di questi ultimi è dunque determinato dalla dimensione del vettore task, che come già sottolienato è di default costituito da 512 elementi. Inoltre per comodità si mantiene una variabile current che punta sempre alla struttura corrispondente al processo correntemente in esecuzione. I campi principali della struttura `task_struct` sono qui di seguito descritti:

- **stato**: variabile che può assumere i consueti valori di *Running*, *Waiting*, *Stopped*,

Zombie

- **informazioni di schedulazione**: utili allo scheduler per decidere quale processo ha maggior bisogno di girare;
- **identificatori**: i noti identificatori dei sistemi Unix, quali il process identifier, e gli user e group identifier, usati in genere per controllare gli accessi al file system;
- **inter-process communication**: usati per i tipici meccanismi di IPC di Unix, quali i segnali, le pipe ed i semafori, e per quelli System V IPC, come la memoria condivisa, i semafori e le code di messaggi;
- **links**: per poter risalire per esempio al padre del processo stesso e a tutti quelli con lo stesso padre;
- **timers**: per tener traccia del momento di creazione, del tempo di CPU consumato e cosìvia;
- **file system**: puntatori ai descrittori dei file aperti;
- **virtual memory**: puntatori alle strutture `vm_area_struct` viste in precedenza;
- **processor specific context**: strutture dipendenti dalla particolare CPU utilizzata, che contengono lo stato in termini di registri, stack e cosìvia.

Analizziamo ora cosa avviene quando si richiamano le due principali system call di gestione dei processi, cioè la `fork()` e la `exec()`.

6.1 La `fork()`

E' noto che nei sistemi operativi di tipo Unix un'operazione fondamentale sui processi è gestita dalla system call `fork()`: essa permette di ottenere una copia identica del processo in cui viene lanciata. L'operazione avviene nel kernel ed in kernel mode. Alla fine viene allocata una nuova `task_struct` ed una o più pagine per lo stack, a cui corrisponde un PID unico nel sistema.

La `task_struct` del nuovo è copiata da quella del vecchio processo, salvo alcuni aggiornamenti, quali assegnare come padre al nuovo processo il PID di quello da cui è stato clonato. In questo procedimento, come già detto nella parte relativa alla memoria, Linux fa in modo che i processi condividano la maggior parte delle risorse possibili, invece di mantenere due copie separate. Questo si applica sia ai file, sia ai signal handler, sia alla memoria virtuale. Linux tiene traccia di ciò incrementando i reference counter delle risorse

condivise, che non dealloca fintantoch entrambi i processi non avranno finito di usarle. Per la memoria virtuale, bisognerebbe generare un nuovo insieme di `vm_area_struct`, e le corrispondenti page table. Niente di tutto ciò è però svolto in questo istante: si adotta infatti la tecnica del "copy on write", che, come già descritto, duplica le strutture necessarie e copia le pagine solamente se il processo cerca di scriverci. Per quanto riguarda il codice, che per sua natura a sola lettura, esso non verrà mai duplicato. Per far funzionare il "copy on write" è indispensabile marcare le pagine nelle page table con l'attributo di sola lettura, e descriverle nella `vm_area_struct` come pagine "copy on write". Come già visto, in caso di scrittura si genera un page fault, che provvede a farne una copia e aggiornare le page table dei due processi.

Per la gestione dei thread, ossia dei cosiddetti processi leggeri, Linux usa lo stesso meccanismo appena descritto: infatti per il sistema operativo essi sono processi a tutti gli effetti. L'unica differenza è che i dati sono ovviamente condivisi anche in scrittura, poich il modello dei thread prevede questa particolarità.

6.2 La `exec()`

Generalmente in Linux, come in Unix, i programmi sono eseguiti tramite una shell. Essa si preoccupa di individuare l'eseguibile richiesto nel file system tramite la ricerca nel path corrente. Una volta trovato, la shell clona se stessa tramite il meccanismo della fork descritto prima e successivamente il processo figlio cosìgenerato sostituisce l'immagine binaria che sta eseguendo (la shell) con quella del file eseguibile appena trovato. Ciò si ottiene tramite il ricorso alla system call `execve()`, che accetta come parametri il pathname completo dell'eseguibile, gli argomenti e l'ambiente.

Un eseguibile può avere diversi formati, tra i quali anche quello di script (ossia un file contenente comandi di shell). I file di codice eseguibile contengono ovviamente del codice macchina e dei dati, insieme ad altre informazioni necessarie a permettere al sistema operativo di caricarli in memoria e di eseguirli. Il formato più diffuso in Linux è l'ELF (Executable and Linkable Format), ma il sistema operativo è progettato in maniera sufficientemente flessibile da eseguire praticamente qualsiasi formato di file (sempre siano state scritte delle apposite routine o esista un interprete come nel caso dei file compilati dal linguaggio Java). Una volta attivato il loader per il particolare formato di eseguibile da caricare, esso tramite la system call `mmap()` crea ed inizializza le `vm_area_struct` corrispondenti alle aree di memoria virtuale e di conseguenza le corrispondenti tabelle di paginazione, ma non copia assolutamente l'immagine del file in memoria. Una volta che il programma viene eseguito, il primo page fault farà in modo che una parte del codice e dei dati siano caricati realmente in memoria: di conseguenza, le parti non usate del programma non verranno mai portate

nella memoria fisica, ottenendo un certo risparmio di risorse. Si realizza cosìl'"on demand loading" di cui si diceva nel capitolo della memoria. Una volta che Linux ha inizializzato la memoria virtuale come richiesto dal nuovo processo, invalida l'immagine eseguibile corrente che era stata clonata dal processo originale tramite la system call `fork()`. Per far ciò, dealloca le vecchie strutture dati della memoria virtuale e cancella le page table. Inoltre elimina tutti i signal handler precedentemente definiti e chiude i file che erano aperti. A questo punto il processo è pronto per eseguire la nuova immagine: non importa che tipo di file eseguibile è stato caricato, perch ora per il funzionamento si utilizzano solo i valori contenuti nella `mm_struct` del processo, quali i puntatori all'inizio del codice eseguibile, all'inizio della sezione dei dati, agli argomenti da passare al processo, alle variabili d'ambiente e cosìvia. Se sono necessarie delle librerie dinamiche, queste sono mappate in memoria virtuale sempre con lo stesso meccanismo, inserendo delle apposite `vm_area_struct` con tutte le informazioni del caso.

Capitolo 7

Considerazioni generali

In questo capitolo si cercherà di rispondere a questa domanda, che forse il lettore si sarà già posto: per quale motivo l'indirizzo base del segmento del kernel è stato posto esattamente all'indirizzo lineare corrispondente a 3 GB, e quali implicazioni questo comporta sull'organizzazione e le prestazioni del sistema?

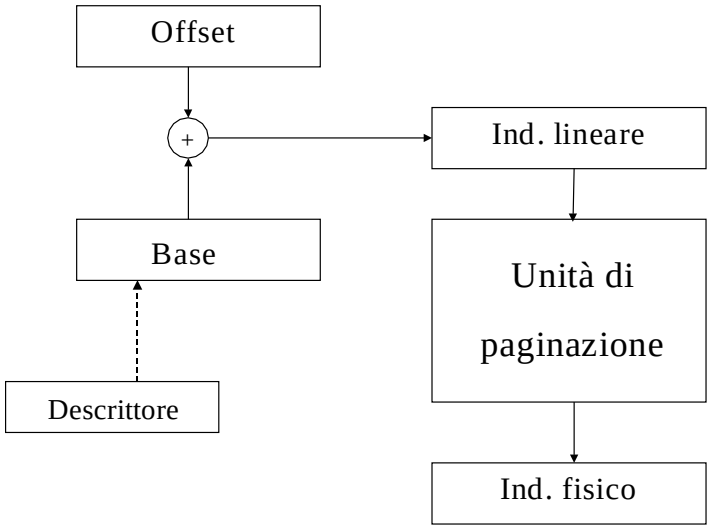

Figura 7.1: Segmentazione e paginazione

Prima di rispondere al quesito, si ricorda brevemente come funzionano i meccanismi di segmentazione e di paginazione sull'architettura x86. E' noto che esistono tre tipi di indirizzi in un'architettura che usi sia la segmentazione sia la paginazione: gli indirizzi virtuali, gli indirizzi intermedi che escono dall'unità di segmentazione (che nell'architettura x86 sono detti lineari) e gli indirizzi fisici. L'indirizzo virtuale è il valore che i programmi

utilizzano per referenziare una variabile in memoria. L'indirizzo lineare, come detto, è quello generato dall'unità di segmentazione, ossia il virtuale a cui è stato sommato il valore base del segmento. L'indirizzo fisico scaturisce dall'analisi delle tabelle di paginazione del sistema: si usa quello lineare come chiave di accesso nelle suddette tabelle e si ottiene quello fisico, che esce dai piedini del processore e va direttamente ad indirizzare la cella nel banco di memoria corrispondente. Il procedimento è illustrato schematicamente nella figura 7.1.

Per il kernel questo meccanismo possiede la particolarità che ad ogni indirizzo virtuale in kernel mode corrisponde il medesimo indirizzo fisico. Per convincersi di questo fatto si può vedere come vengono riempite le tabelle di paginazione nel file arch/i386/kernel/head.S e successivamente in arch/i386/mm/init.c. L'indirizzo virtuale usato dal kernel fa dunque sempre accesso all'indirizzo fisico di memoria con lo stesso valore. Questo è fatto perch è il kernel stesso che deve preoccuparsi di cambiare al volo queste tabelle per un buon funzionamento della memoria virtuale dei processi. Sarebbe piuttosto complicato se non impossibile che il kernel provvedesse a cambiarsi al volo le pagine mentre è in esecuzione; viene perciò stabilita questa corrispondenza biunivoca in fase di avvio, e mantenuta per tutto il tempo di funzionamento del kernel, ossia finch non viene premuto il tasto di reset o viene spenta la macchina.

Poich però le strutture di paginazione sono progettate per mappare indirizzi lineari di massimo 32 bit, non si può andare oltre l'indirizzo lineare corrispondente a 4 GB -1. In pratica, avendo posto la base del segmento del kernel a 3 GB, non è possibile accedere alla memoria fisica al di sopra di 1 GB in modo kernel utilizzando queste tabelle senza modificarle al volo. Questo, unito al fatto che il kernel necessita di accedere direttamente a tutta la memoria fisica per poter espletare pienamente i suoi compiti, limita di fatto la dimensione della memoria fisica utilizzabile a 1 GB (per la verità un po' meno per lasciare spazio a vari periferici che mappano i loro indirizzi di I/O nello spazio di memoria). La figura 7.2 illustra la situazione.

Il rimedio è ovviamente spostare l'indirizzo base del kernel a 2 GB, arrivando cosìa quasi 2 GB di memoria utilizzabile. Si segnala che però l'operazione non è molto semplice nel kernel 2.0, in quanto non esiste un unico punto in cui il valore è stato definito, e forse alcune parti di codice basano il loro funzionamento su questo valore inteso come fisso a 3 GB.

Si noti inoltre che non ha senso abbassare il valore base al di sotto dei 2 GB, perch in questo caso sarebbe lo spazio riservato ai processi a non vedere poter più usufruire di tutta la memoria installata, con evidente inutilità della stessa.

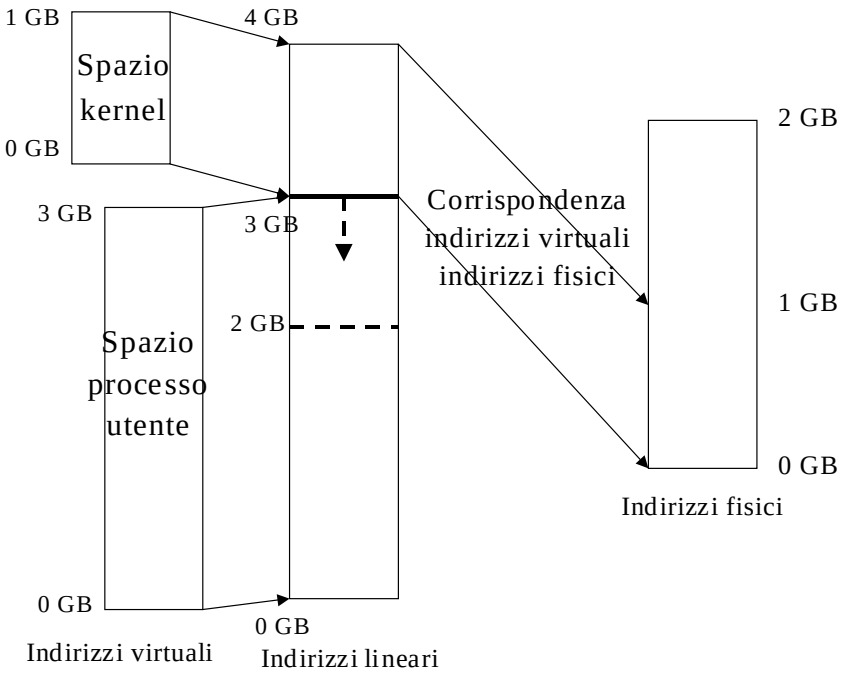

Figura 7.2: Indirizzamento

Capitolo 8

Variazioni con il kernel 2.2

Si danno qui brevi cenni sulle modifiche sostanziali apportate con la nuova versione del kernel, cioè la 2.2, che è uscita all'inizio del 1999. Queste modifiche hanno riguardato, per quanto riguarda la versione per piattaforma Intel, per la maggior parte codice dipendente dall'architettura, in particolare l'organizzazione dei segmenti e il meccanismo del context switching.

Come di consueto, ogni nuova versione aggiunge molti nuovi driver per periferiche e schede di espansione (include infatti tutto quanto si è già riusciti a rendere stabile nell'ambito del kernel sperimentale 2.1).

Vista la sempre maggiore diffusione del sistema, ora Linux 2.2 supporta molte più piattaforme che non la precedente versione. Le architetture correntemente supportate sono ben otto: Intel 386, Alpha, Sparc, Sparc64, Mips, PowerPC, M68000, Arm.

La suddivisione in sottodirectory dei sorgenti descritta all'inizio non è variata, a parte piccoli cambiamenti, se non per aggiunte di nuove funzionalità.

8.1 Segmentazione

I segmenti nell'architettura x86 sono sempre quattro, ma ora sono stati resi tutti coincidenti e di dimensione massima, ossia 4 GB. In pratica si elimina la suddivisione tra kernel e user basata sui segmenti, che viene demandata al kernel stesso tramite la gestione accurata delle pagine. Rimane comunque un punto di separazione tra spazio user e spazio kernel, posto sempre a 3 GB. Ora è però più facile cambiare questo valore grazie ad alcune modifiche apportate ai sorgenti. E' sufficiente sostituire il valore 0xC000000 nel file include/asm/page.h (`#define __PAGE_OFFSET`) e in quello usato dal linker per unire assieme le varie parti del sistema (arch/i386/vmlinux.lds). Ovviamente, per mantenere la separazione tra kernel e

user, il linker provvede a sommare a tutti i riferimenti del codice del kernel il valore 3 GB, per consentire di posizionarlo senza difficoltà nei due segmenti che si dicevano prima, che hanno indirizzo base pari a zero.

8.2 Context switching

Nella versione 2.2 non è più sfruttato il meccanismo hardware dei processori x86. Le routine sostitutive si trovano nei file include/asm/system.h e arch/i386/kernel/process.c.

Nel primo è inserita una routine assembler che pone sullo stack il valore di ESP e di EIP del nuovo processo, ed effettua una jmp all'indirizzo della routine nel secondo file (scritta in C ma con molto assembler inline) che aggiorna i valori del task register del processore con quello proprio del nuovo processo, del Local Descriptor Table Register (se è necessario) e del registro CR3 che contiene l'indirizzo fisico della tabella di paginazione di primo livello. Si noti che il caricamento del task register non provoca il context switching. Esso è però impostato al fine di poter sfruttare la protezione offerta dall'hardware tramite l'I/O bitmap contenuta nel TSS del processo. Tutto il resto del codice funziona come nella vecchia versione: eseguendo la ret della procedura C, il processore carica i valori di ESP ed EIP posti sullo stack in precedenza e prosegue l'esecuzione in kernel mode finch non vi esce tramite l'istruzione iret.

Nelle note che precedono la funzione C (di nome `__switch_to()`), si precisa che questa lunga serie di istruzioni non è assolutamente più lenta del context switching hardware, ed anzi permette una maggiore flessibilità, per esempio nel caso di recupero da valori invalidi dei selettori da caricare, cosa alquanto complessa da implementare sfruttando il meccanismo hardware. Inoltre i problemi del coprocessore vengono risolti in maniera più semplice con una coppia di istruzioni fnsave e fwait, senza l'utilizzo del flag di task switch: ciò semplifica molto le cose nel caso delle architetture multiprocessore e non incide eccessivamente sulle prestazioni.

Capitolo 9

Riferimenti

Poich Linux è principalmente sviluppato da persone in ogni parte del mondo il cui modo di comunicare tra loro è solitamente la rete Internet, è ovviamente a questa che bisogna rivolgersi per ottenere ulteriore documentazione. In particolare si possono visitare i seguenti siti:

http://www.kernel.org dove si possono reperire i sorgenti delle ultime versioni del kernel come di quelle più vecchie. Questi sono ovviamente lo strumento principale che bisogna avere a disposizione per poter dare uno sguardo in dettaglio ai meccanismi interni di Linux.

http://khg.redhat.com dove si può trovare la "Kernel Hacker's Guide", un documento un po' datato e riferito a versioni vecchie del kernel, ma comunque sempre di un certo interesse. Infatti certi meccanismi basilari sono sempre gli stessi che sono già presenti nelle versioni più vecchie del kernel.

http://sunsite.unc.edu/LDP dove si troveranno informazioni di carattere generale su Linux (l'acronimo LDP significa Linux Documentation Project).

Ci è parso utile accludere questa appendice alla relazione in quanto se si vogliono analizzare i sorgenti di Linux, alcune volte ci si imbatte in spezzoni di codice scritti in linguaggio a basso livello, ossia in assembler, ed in particolare con le convenzioni usate dall'assembler AT&T, che differisce un po' dal più abituale assembler Intel.

Appendice A

L'assembler AT&T

Probabilmente si è già notato che ai nomi dei registri è stato preposto un carattere %. Inoltre si noti che l'ordine degli operandi, quando le istruzioni ne supportano due, è invertito rispetto all'assembler Intel: qui l'operando sorgente precede sempre quello di destinazione. Il caso più appariscente è ovviamente quello delle istruzioni mov. Inoltre allo mnemonico di ogni istruzione che può operare su operandi di dimensione differente è aggiunta una lettera che indica la dimensione degli operandi: si avranno cosìle istruzioni movb, movw, movl, che accettano operandi rispettivamente di 8, 16 e 32 bit. Altri esempi sono costituiti da: pushl, popl, stosl, imull.

Inoltre una cosa analoga accade per le istruzioni di salto quali le jmp: per indicare se queste sono di tipo intersegment (ossia FAR nella notazione Intel), si aggiunge una lettera l all'inizio dell'istruzione, ottenendo per esempio dalla jmp la ljmp (già vista nel caso del context switching).

Le costanti numeriche sono indicate tramite un carattere `iniziale`.

La dereferenziazione dei valori, effettuata con le quadre nello standard Intel, qui viene ottenuta tramite le parentesi tonde.

A.1 Le estensioni CYGNUS all'assembler inline

Tutto quanto descritto finora si applica bene ai file sorgente scritti interamente in linguaggio assembler, ma in Linux si fa anche un uso estensivo del codice assembler inline, che segue una notazione leggermente differente da quello proprio dei sorgenti completi, per consentire una migliore ottimizzazione da parte dell'ottimizzatore di codice intermedio. Il codice inline è sempre inserito in statement di questo tipo: asm("statements": output_registers : input_registers : clobbered_registers); dove il contenuto di statements, output_registers e

input_registers è facilmente intuibile, mentre i clobbered_registers sono quelli utilizzati per elaborazioni intermedie.

Un esempio può essere quello per inizializzare un vettore:

```
asm("cld\n\t" \
    "rep\n\t" \
    "stosl"   \
 : /* nessun registro di output */ \
 :"c"(dim), "a" (value), "D" (vector) /* registri
  di ingresso */ \
 :("%ecx", "%edi"); /*registri coinvolti nella rep stosl*/ \
```

Per indicare i registri si usano degli mnemonici, di cui "c", "a", e "D" sono degli esempi. Segue una breve tabella dei più comuni:

a eax b ebx c ecx d edx S esi D edi I costante a 32 bit q registro scelto dal compilatore tra eax, ebx, ecx, edx r registro scelto dal compilatore tra eax, ebx, ecx, edx, esi, edi g registro eax, ebx, ecx, edx o variabile in memoria A eax ed edx insieme per formare un intero a 64 bit

Ovviamente si può anche lasciare la scelta dei registri migliori al compilatore, come si vede per esempio dal codice del context switching:

```
__asm__("movl %2,"SYMBOL_NAME_STR(current_set)"\n\t" \
"ljmp %0\n\t" \
"cmpl %1,"SYMBOL_NAME_STR(last_task_used_math)"\n\t"\
"jne 1f\n\t" \
"clts\n" \
"1:" \
: /* no outputs */ \
:"m" (*(((char *)&next->tss.tr)-4)), \
 "r" (prev), "r" (next)); \
```

dove si vede l'uso delle espressioni quali %0, %1, %2 che si riferiscono rispettivamente alla prima, seconda e terza espressione dopo i due punti che chiudono la sezione degli statement, in questo caso:

%0 è (*(((char *)&next− >tss.tr)−4))
%1 è (prev)
%2 è (next)

Le variabili `prev` e `next` possono essere poste dal compilatore nei registri che ritiene più opportuni (si veda la lettera "r" che li precede).

Per un ulteriore approfondimento si possono ovviamente consultare la esauriente documentazione in formato elettronico acclusa al compilatore nelle distribuzioni di Linux.

Appendice B

Glossario

Call gate Struttura utilizzata da un'istruzione di call in modo protetto quando è necessaria una chiamata ad un flusso di esecuzione non direttamente accedibile per motivi di protezione

CPL (Current Privilege Level) Livello di privilegio corrente del codice

Descrittore Struttura dati di 8 byte utilizzata per descrivere particolari oggetti di sistema, quali i segmenti e i gate nei processori della famiglia x86 in modo protetto

DPL (Descriptor Privilege Level) Livello di privilegio associato da un descrittore ad un oggetto come un segmento o un gate

ELF (Executable and Linkable Format) Il più diffuso formato di file eseguibile sotto Linux.

EPL (Effective Privilege Level) Valore effettivamente utilizzato per determinare l'accessibilità dei dati. Si calcola come $max(CPL, RPL)$

Gate Struttura contenente tutte le informazioni necessarie per la modifica sicura del flusso di esecuzione di un programma

GDT (Global Descriptor Table) Tabella usata dai processori x86 in modo protetto che contiene i descrittori di varie strutture utilizzate dal sistema in generale

GDTR (Global Descriptor Table Register) Registro che contiene l'indirizzo base e il limite della **GDT**

IDT (Interrupt Descriptor Table) Tabella usata dai processori x86 in modo protetto che contiene i descrittori dei gate utilizzati quando si scatena l'interrupt o la trap corrispondente

IDTR (Interrupt Descriptor Table Register) Registro che contiene l'indirizzo base e il limite della **LDT**

IPC (Inter Process Communication) Meccanismo tramite il quale due processi possono comunicare tra loro. Due esempi noti sono i segnali e le pipe

IRQ (Interrupt Request) Segnale utilizzato da un dispositivo periferico per richiedere l'esecuzione di una routine di servizio alla CPU

Indirizzi fisici Indirizzi utilizzati dalla CPU per accedere effettivamente alle corrispondenti locazioni della memoria fisica. Sono equivalenti agli indirizzi lineari se la paginazione non è attiva, altrimenti sono generati dalla unità di paginazione

Indirizzi lineari Indirizzi generati da quelli virtuali tramite il processo di segmentazione

Indirizzi virtuali Indirizzi comunemente utilizzati in fase di scrittura del codice e generati dello stesso in fase di esecuzione

Interrupt Particolare segnale proveniente dall'esterno della CPU che provoca l'interruzione del flusso di codice corrente a favore dell'attivazione di una routine di servizio.

Interrupt gate Struttura di controllo usata quando la CPU riceve un interrupt, per determinare l'indirizzo della routine da eseguire e i relativi privilegi

LDT (Local Descriptor Table) Tabella usata dai processori x86 in modo protetto che contiene i descrittori di varie strutture che possono essere utilizzate dal processo correntemente in esecuzione

LDTR (Local Descriptor Table Register) Registro che contiene il puntatore al descrittore della **LDT** corrente

LILO (Linux Loader) Nome del particolare boot loader di Linux su architettura x86.

Livello di privilegio Valore che indica la possibilità o meno di effettuare operazioni rischiose per l'integrità del sistema. Nei processori Intel questo valore varia da 0 (massimo privilegio) a 3 (minimo privilegio). Linux usa solo 2 livelli: 0 per il kernel e 3 per i processi in modo user.

Page Directory Tabella tramite la quale si può determinare la posizione della Page Table in cui ricercare l'indirizzo fisico corrispondente all'indirizzo assegnato (nei processori x86 può contenere fino a 1024 entry ed è ampia 4 KB)

Page Table Tabella che contiene l'indirizzo fisico di alcune pagine in memoria (nei processori x86 può contenere 1024 entry ed è ampia 4 KB)

Pagina Unità elementare di memoria per i processori che usano la paginazione. Sui processori della famiglia x86 essa è ampia 4 KB.

Paginazione Meccanismo tramite il quale si passa da un indirizzo virtuale ad un indirizzo lineare

PID (Process Identifier) Valore utilizzato dal kernel per identificare un processo. Esso è unico in tutto il sistema e viene assegnato al momento della creazione.

Processo Istanza di un programma nel sistema. Esso è identificato univocamente nel sistema da un PID.

RPL (Register Privilege Level) Occupa gli ultimi due bit di un registro selettore. Serve per il restringimento del livello di protezione

Script File di testo contenente un insieme di comandi di shell

Segmentazione Meccanismo tramite il quale si passa da un indirizzo lineare ad un indirizzo fisico nella memoria; ciò è ottenuto tramite la Page Directory e la Page Table

Segmento Unità logica elementare di memoria. E' in genere caratterizzato da un indirizzo base e da un limite, oltre che da alcuni bit che ne definiscono le caratteristiche ed i relativi permessi

Selettore Registro dei processori x86 utilizzato per referenziare un particolare segmento di memoria

Shell Particolare programma che si occupa di eseguire i comandi impartiti dall'utente.

Task Processo del sistema operativo. Sinonimo di processo.

Task Gate Gate tramite il quale si può ottenere un context switching

Thread Processo leggero, che condivide con il padre alcune risorse generalmente non condivise, quali per esempio la memoria

TLB (Translation Lookaside Buffer) Cache usata dall'unità di paginazione per la traduzione da indirizzi lineari a indirizzi fisici

TR (Task Register) Registro che contiene il puntatore al descrittore del TSS corrente

Trap Interrupt sincrono generato internamente alla CPU. Detta anche eccezione

Trap gate Struttura di controllo usata quando la CPU genera una trap, per determinare l'indirizzo della routine da eseguire e i relativi privilegi

TSS (Task State Segment) Area di memoria usata dai processori x86 in modo protetto per salvare e ripristinare lo stato di un task precedentemente interrotto. Può contenere inoltre altre informazioni riguardanti il task quali la I/O bitmap.