Ottimizzazione zero-fill-on-demand

Idea base: vengono allocate le nuove pagine come richiesta

Si potrebbero, dopo una richiesta dello spazio, creare, bene appena prima del processo.

Si collegano pagine "vuote" ovvero con byte nulli

Il pezzo di memoria che diamo ora è stato un precedente proprietario, dunque il pezzo è "sporco"

Data questa proprietà si deve gestire

- azzeramento (per non violare l'astrazione del "virtuale")
- sicurezza (Es: se la password di un qualcosa era in quel pezzo?)

Trasformare una pagina in vuoto ha un costo notevole.

Il SO gestisce il tutto con due diverse tecniche

- pool di pagine vuote (rozza)
- Copy-on-write su una read only static zero page

In questa tecnica una pagina fisica viene sempre tenuta vuota e usata come "stampo" per eventuali pagine vuote

E' una pagina condivisa da tutti i processi

Sostanzialmente il processo avrà molte pagine che

fanno riferimento alla pagina vuota.
Quando provano a scrivervi accade la copy on write
Isolando una frame libera dalla RAM su cui è stata copiata la pagina.
Ovviamente supponendo ci siano frame libere, altrimenti scatta lo swapping
L'allocazione dunque nel caso ci siano frame libere diventa più semplice.
L'altra tecnica consiste nel tenere una pool di frame libere VUOTE. Si pulisce [zero] nei momenti in cui non sta venendo fatto altro lavoro e si sfrutta il meccanismo di DMA
È un compito affidato in WINDOWS 8 a un processo con priorità 0 (minima)

Librerie condivise e file mappati

Più processi tendono ad usare porzioni di codice che stanno nelle librerie
Come si "tocca" il codice della libreria?
- linking statico: si include il codice (quello che serve) in fase di linking (pre compilazione)
- linking dinamico: collegamento e caricamento a è il più diffuso run time
Solitamente si ritarda più possibile il linking
A run time viene caricata la libreria in memoria
Le librerie andrebbero ovviamente gestite in sola lettura
Se una libreria è caricata in RAM, è vista da tutti i processi in condivisione (si evita di caricarla in RAM tante volte)
Ogni processo usa puntatori alle routine della libreria
- Si risparmia spazio
- Sviluppo e aggiornamento semplificati.

File mappati

La mappatura migliora le prestazioni e consente comunicazione tra processi

- modello di interazione con i file (è un modello alternativo al classico I/O)

Idea: mappare il contenuto del file nello spazio di indirizzamento (nella RAM poi potrà anche non esserci contiguità)

Dal punto di vista logico viene presa una porzione dello spazio di indirizzamento e fatta coincidere col file

Mappatura ≠ caricamento

Il caricamento è rimandato.

Appena il processo prova a fare una fetch il SO provvede col caricamento in RAM del blocco del file che il processo ha provato a usare.

Con questa tecnica stiamo rimandando il più possibile l'I/O

La logica è applicata anche in scrittura, se il processo prova a modificare il contenuto all'inizio la modifica avverrà in RAM e poi avverrà in seguito la sincronizzazione tra la copia in RAM e la copia in Disco.

Con questa tecnica si possono gestire file enormi di fatto usando pochi pezzi

In questo modello esiste la possibilità di condivisione il blocco del file può essere messo in una frame condivisa

Il concetto di mappatura gestisce automaticamente:

- librerie condivise (pezzi di area mappati in memoria)
- caricamento codice eseguibile
- caricamento dati statici

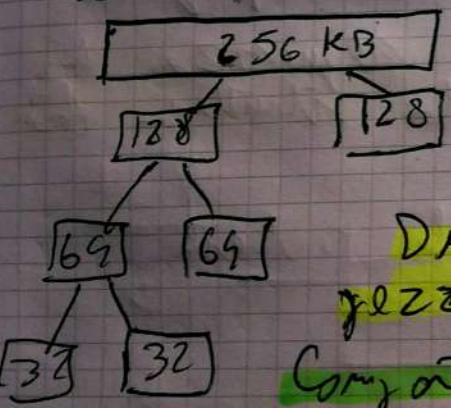
Se questi campi non prevedono scrittura, due o più processi potrebbero condividere le pagine relative

Allocazione memoria per il Kernel

Come è gestito lo spazio di lavoro del SO?
Problema dell'uovo e della gallina: se il SO implementa le astrazioni, già a una volta usare le astrazioni?
Sì e no
La paginazione non può essere fatta per qualunque compito del SO: troppo Overhead, non può usare un supporto se lo sta implementando il supporto
Nella pratica il SO moderno ha un approccio misto ovvero che prevede memoria contigua con qualche accorgimento per ottimizzazione
Mentre nell'allocare i processi utente usiamo la paginazione che ci porta frammentazione interna, nell'allocare processi del Kernel puntiamo a frammentazione interna minima.

La soluzione

Buddy system: gestire uno spazio contiguo e molto per richieste del SO. Di solito è un pezzo con una dimensione pari a una potenza di 2
Questo spazio è usato per le richieste che arrivano dal SO stesso, ogni richiesta ha una sua taglia



Se arriva una richiesta di 25 KB, saltiamo alla potenza di 2 sufficientemente grande i 32
Dunque splittiamo lo spazio per dargli il pezzo più piccolo possibile
Comporta sprechi (frammentazione interna) ma è semplice e permette coalescenza delle richieste quando libera la "chiave"
Si può creare anche frammentazione esterna

Slab allocator 2ª soluzione

- no greco
- efficiente

Permette di gestire tramite pezzi di RAM le richieste del SO.
Una richiesta del SO è più analizzare a quale e individuare che tipo di struttura dati è opportuna.
Della richiesta ciò che importa è la dimensione
Si stabiliscono a tavolino le strutture dati utili per il SO in base alle sue richieste
Posso immaginare ad esempio che il mio SO usi 3 tipi di strutture: 1KB, 3KB, 12KB
Per ognuna di queste dim è creata una cache (è una struttura dati che gestisce slab)

Slab: sequenza di frame (contigue)

Come può uno slab appartenere a una cache?

- frame contigui
- la sua dim è un multiplo della dimensione chiave della cache e multiplo della dim della pagina

Uno slab può essere visto come una sequenza di slot idonei a contenere copie della struttura dati relativa alla cache in uso
Lo slab è usato per soddisfare le richieste del Kernel (che hanno dim fisse)

Come è usato lo slab?

Se mi interessa uno idoneo e poi ne uso un tassello di quello slab per la richiesta

- Non c'è frammentazione

La gestione dello slab prevede lo stato: vuoto / pieno / parziale
La gestione è dinamica: una cache può aumentare o diminuire gli slab disponibili

Lo slab allocator è applicabile nel kernel in quanto lavora a livello delle zone. È inutilizzabile per i processi utente

Segmentazione

Come vede l'utente la memoria?

- array lineare di byte
- collezione di oggetti con dim variabile

Alternativa: segmentazione

- Viene eseguita la logica visione dell'utente
- si crea un segmento per ogni oggetto, procedura, stack, libreria.

Segmentiamo la memoria per questi oggetti
Ogni segmento contiene dei dati omogenei, relativi allo stesso "concetto"
Ogni segmento ha dimensione variabile.

Dunque la memoria è vista come una collezione di segmenti dove ogni segmento è una collezione di dati omogenei

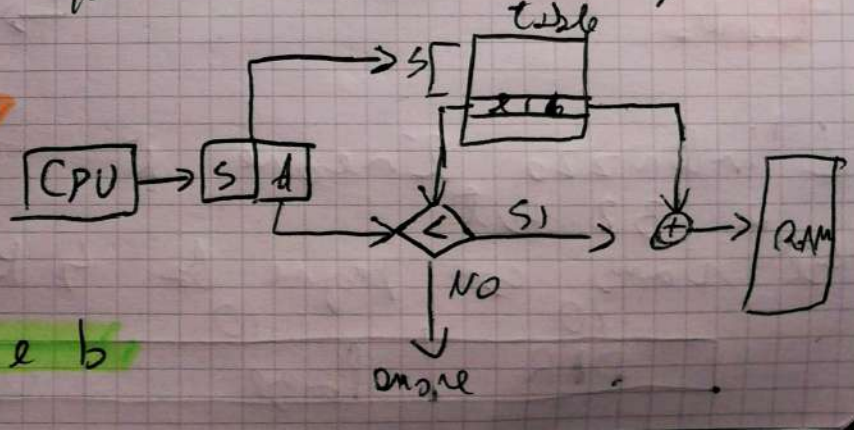
Una word è adesso individuata da una coppia di valori, detta indirizzo bidimensionale

- indirizzo bidimensionale: (#segmento, offset)
- tabella segmenti: per la traduzione in i.f.

Permette di separare le informazioni in modo logico e puro

Vediamo uno schema per la traduzione e controllo dell'indirizzo bidimensionale in fisico

Ogni segmento ha #S, l e b



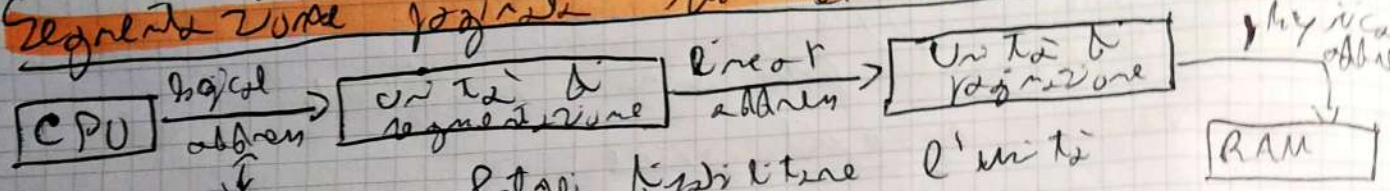
La Tabella di occupazione della memoria tramite segmentazione

segmento → (base, limite)

Problema: frammentazione esterna

Soluzione: sistema ibrido Paginazione + Segmentazione

"Segmentazione paginata" nei Pentium Intel 32 bit



indirizzo bidimensionale

Potrei disabilitare l'unità di seg o di paging.

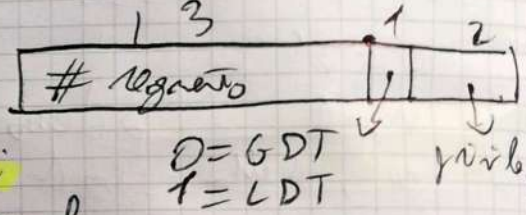
Ci sono circa 16000 segmenti a 4 GB (32 bit)

Usiamo due tabelle dei segmenti (8000 circa ciascuna)

- LDT: segmenti per i programmi (uno per ogni processo)
- GDT: segmenti del SO (globale)

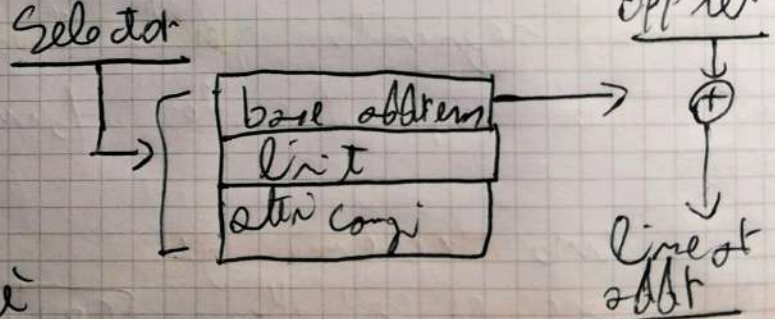
Usiamo 6 registri per la gestione dei segmenti, sono detti selettori perché servono appunto a selezionare segmenti specifici

I selettori sono a 16 bit:



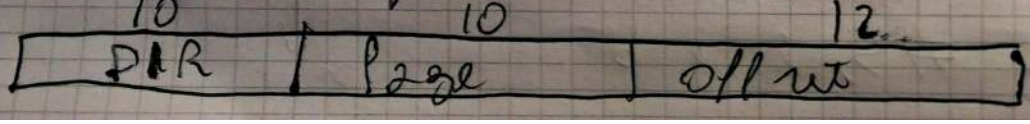
I Pentium supportano 4 livelli di privilegi, non solo User e Kernel mode

Come ottenere il linear addr dalla tabella

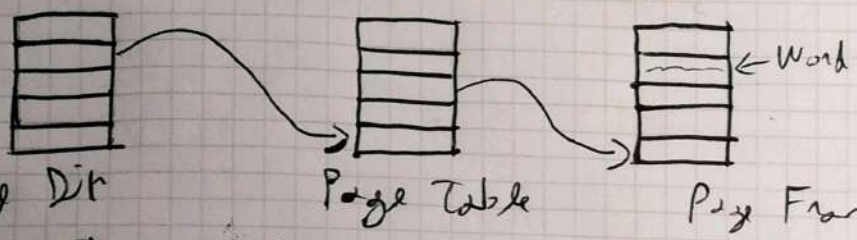


L'indirizzo lineare è a 32 bit

Se usiamo con pagine da 4 KB



Nella gestione si usa una Tabella multilivello dove:
- PT1 = DIR
- PT2 = Page

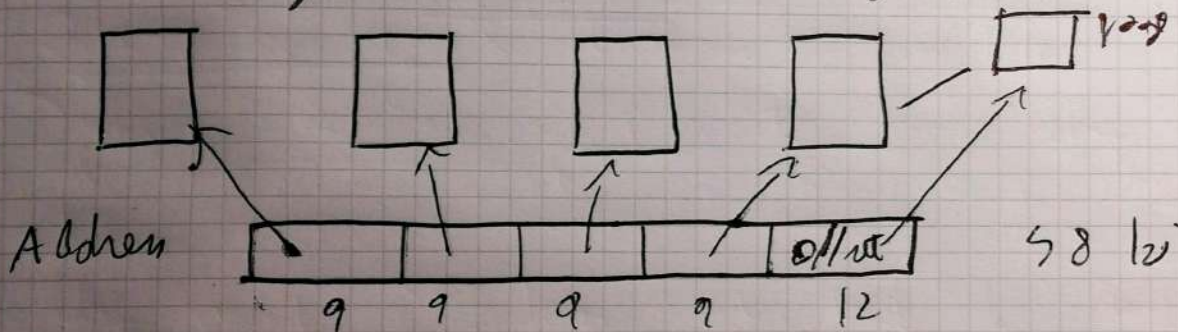


E' una Tabella a 2 livelli

In questo sistema è la swapping solo nella tabella di 2° livello

Usa la TLB

Nelle architetture x86-64 abbiamo una Tabella a 4 livelli e si limita la segmentazione
Qui gli indirizzi virtuali effettivamente generati sono a 48 bit (parte alta nulla o ignorata)
Per la gestione dei livelli della tabella



La segmentazione è limitata e si usano solo 2 livelli di protezione

limitata all'uso nel SO

La memoria in LINUX è gestita con
- copy on write e read only static zero page
- file mappati (codice e libreria condivise)
- allocazione dinamica Heap usando brk() (syscall)
- algoritmo sostituzione pagine: ibrido tra Clock e NRU
- slab allocation per la memoria riservata al Kernel