La Bibbia di Sistemi operativi

Mario Petruccelli
Università degli studi di Milano

A.A. 2018/2019

Sommario

1 Virtualization

1.1 Introduzione

Processi Un processo, informalmente, è un programma in esecuzione. Un programma a sua volta, è una sequenza finita di istruzioni scritte in un linguaggio comprensibile all'esecutore (CPU). L'esecuzione di un programma da parte del processore è:

- **Fetch** Prelievo istruzione dalla memoria.
- **Decode** Decodifica dell'istruzione.
- **Execute** Esecuzione dell'istruzione.

1.1.1 Virtualizzazione

La virtualizzazione consiste nel prendere una risorsa fisica e trasformarla in una più generale, potente e facile da adoperare forma virtuale di se stessa.

Virtualizzazione della CPU L'illusione consiste nel far credere che il sistema abbia un elevato numero di cpu virtuali. Avere più CPU permeterrebbe a più programmi di essere eseguiti in **parallelo** nonostante il processore fisico effettivo sia uno solo. Se due processi vogliono essere eseguiti entrambi ad un certo tempo, oppure vogliono accedere alla stessa periferica, quale dei due ha la priorità? La risposta viene data con l'introduzione delle politiche di priorità (**politiche di scheduling**).

Virtualizzazione della memoria Consiste nel fabbricare l'illusione che ogni processo abbia il proprio spazio di indirizzi virtuali privato (**address space**) al quale accede e sarà il sistema operativo ad occuparsi di mappare nella memoria fisica.

Con la virtualizzazione è fondamentale riuscire a distinguere i processi in esecuzione. Per fare ciò viene associato un **PID** (process id) ad ogni job. il PID è un numero univoco.

1.1.2 Concorrenza

Si riferisce a tutta quelle serie di problemi che sorgono, e che vanno risolti, quando all'interno dello stesso programma più entità lavorano in parallelo. Le entità in questione si chiamano **threads**.

1.1.3 Persistenza

La persistenza è legata alla memorizzazione dei dati all'interno della memoria. La non volatilità delle memorie ha introdotto la possibilità di memorizzare dati in modo persistente. Il software nel sistema operativo che generalmente gestisce i dischi è chiamato **file system**.

1.1.4 Protezione ad anelli

Un modello di potezione implementato dal sistema operativo è quello ad anelli. Ci sono 5 livelli e 3 anelli differenti. A ciascun anello corrisponde un relativo livello di sicurezza.

- **Level 1** *Hardware level* qui vengono eseguiti, ad esempio, i device drivers visto che essi richiedono accesso diretto all'hardware dei dispositivi (microcontroller).
- **Level 2** *Firmware level* Il firmware sta in cima al livello elettronico. Contiene in software necessario dal dispositivo hardware e dal microcontroller.
- **Level 3: ring 0** *Kernel level* Questo è il livello dove opera il kernel, dopo la fase di bootload siamo qui.
- **Level 4: ring 1 e 2** *Device drivers* I device drivers passano atraverso il kernel per accedere all'hardware.
- **Level 5: ring 3** *Application level* Qui è dove viene eseguito normalemente il codice utente.

1.2 Processi

Sistema multiprogrammato Sistema nel quale è possibile eseguire più programmi contemporaneamente, idea alla base della virtualizzazione.

1.2.1 Multiprogrammazione

Time sharing prevede che il tempo di CPU sia equamente diviso fra i programmi in memoria.

Real time sharing La politica di scheduling è differente. Alcuni processi vanno serviti prima di altri.

1.2.2 Virtualizzazione della CPU

L'illusione consiste nel rendere indipendenti il numero di processi dal numero di processori. Si vuole disaccoppiare le entità logiche (*processi*), dalle entità fisiche (*processori*), in modo tale che ad ogni processo venga assegnato un processore logico mappato su processore fisico.

I concetti fondamentali alla base della virtualizzazione sono:

- **Time sharing** Meccanismo mediante il quale il tempo di CPU viene diviso equamente fra i processi.
- **Context switch** Meccanismo che consente di interrompere l'esecuzione di un processo in corso sulla CPU fisica e assegnare quest'ultima ad un nuovo processo.

1.2.3 Processi

Un processo è un programma in esecuzione, il sistema operativo deve fornire alcune interfacce (**APIs**) per la gestione dei processi che permettano di fare:

- **Create** Creazione di un nuovo processo.
- **Destroy** Eliminazione forzata di un processo. Molti processi termineranno per conto loro, ma l'utente potrebbe voler eliminare processi non ancora terminati.
- **Wait** Mette in attesa un processo.
- **Miscellaneous control** Sospensione di un processo per farlo ripartire dopo un certo tempo.
- **Status** Interfacce che restituiscono lo stato e altre informazioni di un processo.

Creazione di un processo La prima cosa che deve fare il sistema operativo per eseguire un programma è caricare il suo codice ed eventuali dati statici da disco a memoria, nell'address space del processo.

- **Allocazione dello stack** Un po' di memoria deve essere creata per lo stack del programma (*variabili locali, parametri delle funzioni e indirizzi di ritorno*).
- **Allocazione dello heap** Un po' di memoria deve essere creata per lo heap del programma (*dati allocati dinamicamente*).
- **Inizializzazione I/O** Standard input, output ed error.
- **Salto ed esecuzione** Salto all'entry point ed esecuzione. (*main*)

Stato di un processo

- **Running** È in esecuzione sul processore.
- **Ready** In attesa di essere eseguito dal processore.
- **Blocked** In stato di block, il processo sta esegundo qualche operazione (*es: I/O*).

Strutture dati Il sistema operativo deve tenere traccia delle informazioni fondamentali di un processo per poter ripristinare l'esecuzione di un processo interrotto. Esse sono:

- Porzioni di memoria coinvolte.
- Valori dei registri di CPU usati dal processo.
- Stato dei dispositivi di I/O usati dal processo.

Questi dati sono organizzati in strutture chiamate **Process Control Block (PCB)**, salvate in un per-process **kernel stack**, il quale risiede nel kernel space.

1.2.4 Process API

La creazione di un processo avviene tramite la `fork()`, la quale genera un processo identico a quello in esecuzione. Tale processo prende il nome di padre, quello generato viene chiamato figlio. L'esecuzione del processo figlio parte dall'istruzione successiva alla `fork()`. La `fork()` ritorna al figlio 0, al padre il **PID** del figlio e **-1** in caso di errore. Il processo figlio avrà il **proprio** address space, registri, PC, ecc...

La `wait()` è una funzione che forza il padre ad aspettare che il processo figlio termini la propria esecuzione. Senza, l'output potrebbe essere **non-deterministico** e potrebbero crearsi processi orfani o zombie. Esiste anche la `waitpid` che viene usata se si ha a che fare più di un figlio.

Per eliminare un processo esiste la funzione `kill()`. Solo il padre può distruggere il figlio. Ciò può portare alla creazione di processi **zombie** (processi terminati la cui **PCB** è ancora in memoria).

La `exec()` serve per generare un processo che fa qualcosa di diverso da quello padre. `exec(nome_programma, arg)` prende il nome di un eseguibile e alcuni argomenti, carica il codice e i dati statici di quell'eseguibile, sovrascrivendo il code segment corrente all'interno del PCB del figlio. Heap, stack e altre parti di memoria vengono re-inizializzate. Rimane la relazione padre-figlio.

1.3 Context Switch

1.3.1 Shell

Come mai `fork()` ed `exec()` sono due system call separate? Per rispondere introduciamo la **shell**.

La shell è un programma del sistema operativo `Unix` il cui compito è riconoscere ed eseguire altri programmi; si può dire che essa sia il genitore di tutti i processi che vengono mandati in esecuzione. Nello specifico, essa esegue una `fork()`, cambia il file descriptor se richiesto, ed infine invoca la `exec()`. Poi si mette in attesa che il programma abbia terminato prima di tornare in attesa di istruzioni. Esistono due tipi di shell, grafica (*terminale*) e interattiva (*aprire programmi col mouse*).

La separazione di `fork()` ed `exec()` è dovuta alla presenza della shell, con la quale possiamo andare ad effettuare alcune modifiche dopo la `fork()` e prima dell'`exec()`, come ad esempio la sostituzione del file descriptor.

```
$> wc file.c > n.txt
```

La shell esegue la `fork()` per poter mandare in esecuzione il programma `wc`. Prima di sostituire il codice del padre all'interno del PCB del figlio, sostituisce il file descriptor relativo allo standard output con n.txt. Successivamente esegue l'`exec()` producendo l'output desiderato all'interno di n.txt. Queste manipolazioni non sarebbero possibili se `fork()` ed `exec()` fossero un'unica system call perchè non si avrebbe accesso al PCB del figlio prima dell'exec().

1.3.2 Direct execution

Il concetto di direct execution è semplice: il programma viene eseguito direttamente sulla CPU fisica.

Quando il sistema operativo desidera iniziare l'esecuzione di un programma, viene fatto quanto segue:

- Crea una entry nella lista dei processi.
- Alloca la memoria per il programma.
- Carica il programma in memoria.
- Imposta lo stack con `argc/argv`.
- Pulisce i registri.
- Esegue la chiamata a `main()`.
  Si ha un salto dalla zona kernel al `main`. Il processo a questo punto deve:
- Eseguire il codice del `main()`.
- Ritornare dal main a fine esecuzione.
  Dal processo si torna alla zona kernel. Il sistema operativo infine:
- Rimuove la entry dalla lista dei processi.

Tuttavia la direct execution solleva alcune problematiche:

- Il sistema operativo non può assicurarsi che un programma in esecuzione non faccia qualcosa che non dovrebbe fare.
- Il sistema operativo non può fermare un processo in esecuzione.

Il primo problema si risolve con l'introduzione dello **user mode**. Il codice che viene eseguito in questa modalità di elaborazione è limitato in termini di istruzioni eseguibili. Nasce quindi anche la **kernel mode**, modalità in cui opera il sistema operativo e che consente di eseguire tutte le istruzioni privilegiate.

Per permettere ad un processo di eseguire istruzioni privilegiate vengono introdotte delle **system call**. Per eseguirle, un programa deve eseguire un'istruzione **trap** (*interrupt via software*). Questa istruzione salta nel kernel, aumenta i privilegi a kernel mode, esegue le operazioni privilegiate e ritorna al processo scalando i privilegi tramite un'istruzione **return-from-trap**. Durante questo procedimento bisogna assicurarsi di salvare i registri del chiamante. Per sapere dove la trap deve saltare, il kernel imposta una **trap table** al boot time. Non è il processo utente a specificare l'indirizzo dei **trap handlers** perchè potrebbe saltare ovunque nel sistema.
Per specificare la system call, generalmente viene assegnato un **system-call-number** che solitamente viene inserito in un registro appropriato.

1.3.3 Switch tra processi

Cooperative approach Soluzione via software che consiste nel programmare il processo in modo che, dopo un certo numero di secondi di utilizzo della CPU, il comando torni al sistema operativo. Il problema è che se vengono creati loop infiniti nel programma, la CPU non verrebbe mai condivisa.

Time interrupt Soluzione via hardware che consiste nel creare una nuova componente che genera un segnale elettrico (**time interrupt**) dopo un certo lasso di tempo. Ci sarà quindi un orologio interno che invierà un segnale al piedino del microprocessore. L'hardware deve inoltre fermare l'esecuzione del processo corrente, salvarne lo stato per dare il controllo allo **scheduler**, che nel caso decidesse di cambiare processo, farà eseguire al sistema operativo codice a basso livello che prende il nome di **context switch**.

Context switch Ciò che deve fare il sistema operativo è salvare alcuni valori dei registri per il processo in corso di esecuzione (*nel kernel stack*) e ripristinarne altri per il processo scelto. Viene eseguita una return-from-trap per mandare in esecuzione il processo scelto.
Interrupt, system call ed eccezioni sono eventi che inducono il mode switch.

1.4 Scheduling policy

Dati n processi, a quale assegno il processore? La scelta è fatta dallo **scheduler**, un modulo del sistema operativo che implementa una politica decisionale.

CPU burst è l'intervallo di tempo in cui viene usata intensamente la CPU.

I/O burst è l'intervallo di tempo in cui viene usato intensamente I/O.

CPU bound processi con CPU burst lunghi, ad esempio compilatori, simulatori, calcolo del tempo, ecc...

I/O Bound processi con I/O burst lunghi, ciò comporta maggiore interattività con l'utente.

Stato di IDLE è lo stato in cui è una risorsa accesa e funzionante ma non utilizzata.

Un processo in esecuzione si trova o in CPU burst o in I/O burst. Lo scheduler, per essere efficiente, deve ottimizzare l'uso delle risorse in modo tale che, se la CPU è occupata con l'esecuzione di un processo, i dispositivi di I/O lo sono con un altro e viceversa. L'ottimizzazione della CPU viene dunque portata mediante lo scheduler. Per valutare la bontà di un algoritmo di scheduling si devono introdurre delle metriche di valutazione.

$$T_{turnaround} = T_{termine} - T_{arrivo}$$

$$T_{response} = T_{first-exec} - T_{arrivo}$$

$$T_{wait} = T_{turnaround} - T_{job}$$

1.4.1 Algoritmo FIFO

L'algoritmo FIFO (*First In First Out*) mette in esecuzione il primo processo arrivato. Il problema a cui può portare questo algoritmo è l'**effetto convoglio**, ovvero quando un certo numero di piccoli consumatori di una risorsa vengono messi in coda dietro un enorme consumatore.

1.4.2 Algoritmo SJF

L'algoritmo SJF (*Shortest Job First*) mette in esecuzione il processo con CPU burst minore. In questo modo si evita l'effetto convoglio, ma solo se i processi arrivano allo stesso istante.

1.4.3 Algoritmo STCF

L'algoritmo STCF (*Shortest Time to Completion First*) ogni volta che arriva un processo, lo compara il processo in esecuzione e lascia il processore a quello che ha CPU burst minore.

SJF e STCF funzionano molto male per quanto riguarda il tempo di risposta (spesso possono indurre anche al verificarsi della starvation). Inoltre non conoscono a priori il CPU burst di un processo, perciò sono solo algoritmi teorici.

1.4.4 Round Robin

L'algoritmo **Round Robin** assegna un **quanto di tempo** ad ogni processo. Viene inizializzato un timer che, una volta arrivato a zero, forza un context switch. Il quanto di tempo va scelto bene, altrimenti si hanno troppi context switch se è troppo piccolo, o degenera in FIFO se è troppo grande.

1.5 Multilevel feedback scheduler

Il problema che **MLFQ** (*MultiLevel Feedback Queue*) cerca di risolvere è:

- Ottimizzare il $T_{turnaround}$.
- Aumentare l'interattività utente/sistema, minimizzando il $T_{response}$.

L'approccio che si usa consiste nell'avere un certo numero di **code** distinte, ognuna assegnata ad un diverso **livello di priorità**. MLFQ sfrutta i diversi livelli di priorità per decidere quale processo eseguire: viene scelto quello all'interno della coda di priorità maggiore. Se ci sono più processi all'interno di una certa coda, viene usato **RR**.

- 1. If priority(A) > priority(B), A runs (B doesn't).
- 2. If priority(A) = priority(B), A & B run in RR.
- 3. Quando un processo entra nel sistema, viene posizionato nella coda di priorità massima.
- 4a. Se un processo utilizza tutto il lasso di tempo a disposizione durante l'esecuzione, la sua priorità viene ridotta.
- 4b. Se un processo libera la CPU prima di terminare il lasso di tempo a disposizione, il livello di priorità rimane invariato.
- 5. **Priority boost** Dopo un certo periodo di tempo, tutti i processi vengono spostati nella coda di priorità più alta. (*Evita la starvation dei long running jobs e il monopolio della CPU se qualche processo la rilascia poco prima del lasso di tempo.*)

1.5.1 Better accounting

La scelta del tempo è cruciale, se settato troppo grande, i long running jobs potrebbero ancora andare in starvation, se impostato troppo piccolo, i processi interattivi potrebbero non avere una porzione adeguata dela CPU. Per evitare che possa essere raggirato l'algoritmo di scheduling, lo scheduler tiene tracia di quanto tempo ha consumato un processo in un certo livello di **MLFQ**. Le regole 4a e 4b diventano:

- 4. Una volta che un processo ha usato il tempo a disposizione in un certo livello (indipendentemente da quante volte ha rilasciato la CPU), la sua priorità viene ridotta.

1.6 Address space

Un programma per essere eseguito deve risiedere in memoria. Essa può essere usata implicitamente (**stack:** `int x`) o esplicitamente (**heap:** `int *x = malloc(sizeof(int))`). Lo stack è gestito autonomamente, lo heap è gestito dal programmatore attraverso opportune funzioni (`malloc, realloc, free, ...`), per cui non si conosce a priori la dimensione.

1.6.1 Memory API

- `malloc()` Riceve in input un argomento di tipo `size_t` (numero di bytes), se ha successo restituisce un puntatore all'inizio della zona allocata nello **heap**, se fallisce restituisce NULL.
- `free()` Riceve in input un puntatore, la grandezza della regione da liberare viene tenuta nella libreria `memoryallocation`.

La system call per la gestione diretta della memoria è `int brk(void *addr)`

1.6.2 Memory errors

- **Dimenticarsi di allocare la memoria.** È da patchare il fatto che si possa indurre un `segfault` in modo tale da poter accedere al core dump della memoria e vedere dati sensibili (**attacchi core-dump**). È necessario eliminare questi dati dopo il loro utilizzo.
- **Non allocare abbastanza memoria.** Può portare a vulnerabilità come il **buffer overflow**.
- **Dimenticarsi di inizializzare memoria allocata.** Potrebbero esserci valori come 0 o valori random.
- **Dimenticarsi di liberare la memoria.** Il **memory leak** può portare ad un esaurimento della memoria disponibile.
- **Liberare la memoria prima di aver finito di usarla.** Questo errore è chiamato **dangling pointer**, può causare un crash o la sovrascrittura di memoria valida.

- **Liberare la memoria più di una volta.** Problema noto come **double free**, il risultato è indefinito, la libreria `memory-allocation` potrebbe confondersi e fare cose strane. I crash sono la cosa più comune.
- **Chiamata di `free()` incorretta.** La funzione si aspetta un puntatore prodotto in precedenza da una `malloc()`. Quando viene passato alla `free` un valore diverso, possono succedere cose brutte e pericolose.

1.6.3 Virtualizzazione della memoria

Con l'avvento della **multiprogrammazione** la memoria diviene una risorsa condivisa, bisogna iniziare a far fronte a tutte le problematiche che ciò comporta.

- **Protezione** un processo non può invadere lo spazio di un altro.
- **Interattività** Ci devono essere molti processi in esecuzione.

Il meccanismo di astrazione che si vuole implementare prende il nome di **address space**, esso è il punto di vista di un processo sulla memoria del sistema, ovvero l'astrazione che il sistema operativo gli fornisce.

Gli obiettivi della virtualizzazione della memoria sono riassunti come segue:

- **Trasparenza.** Il programmatore scrive il codice indipendentemente dalla grandezza della memoria.
- **Efficienza.** Il meccanismo di virtualizzazione non deve avere overhead troppo elevato.
- **Protezione.** Bisogna proteggere i processi da altri processi, dal sistema operativo, e viceversa.

1.6.4 Mapping

Il **mapping** consiste nel trovare una corrispondenza fra indirizzo logico e indirizzo fisico. Nei sistemi **monoprogrammati** ciò era facile poichè ogni programma veniva mappato a partire dall'indirizzo `64KB` fino alla fine. Il compilatore assegnava ai programmi indirizzi costanti. Nel caso della **multiprogrammazione** invece, il compilatore assegna indirizzi preliminarli al programma, i quali vengono successivamente rilocati.

1.6.5 Base e Bound

Questa tecnica di mapping utilizza due registri, **base** e **bound**. Assunzioni:

- Il programma viene caricato in locazioni contigue di memoria. (Un programma da `32KB` verrà caricato in `32KB` locazioni adiacenti)
- L'indirizzo logico è sempre minore dell'indirizzo fisico.

Mediante la rilocazione siamo in grado di calcolare l'indirizzo fisico come segue:

```
indirizzo fisico = indirizzo logico + Base
```

Base è un registro contenente il punto di partenza (indirizzo fisico) del programma. **Bound** è il registro limite. Se un processo prova a saltare in zone di un altro processo viene generato un errore di segmentazione.

1.6.6 MMU

MMU sta per **Memory Managment Unit** ed è una componente hardware per la rilocazione degli indirizzi. L'input è un indirizzo logico prodotto dalla CPU, l'output è l'indirizzo fisico. Generalmente questa traduzione viene fatta a runtime. Prima di eseguire l'istruzione a cui sto puntando, l'indirizzo logico viene tradotto in indirizzo fisico (**rilocazione dinamica**).
Ore che abbiamo la rilocazione dinamica, il sistema operativo deve fare le seguenti cose per implementare la memoria virtuale:

- Quando un nuovo processo viene creato, il sistema operativo dovrà cercare in una struttura dati (spesso chiamata **free list**) spazio libero per il nuovo address space e marcarlo come in uso.
- Quando un processo termina, deve riabilitare tutta la memoria allocata per il processo all'interno della free list e pulire ogni struttura dati associata ad esso.
- Quando avviene un context switch deve salvare nel PCB i registri base e bound e ripristinare quelli del nuovo processo.
- Quando un processo viene fermato è possibile muovere un address space da una locazione di memoria a un'altra. Basta deschedularlo, copiare l'address space dalla locazione corrente a quella nuova e infine aggiornare il registro **base**.

Il sistema operativo deve fornire degli **exception handler**. Per esempio, se un processo prova ad accedere a memoria al di fuori del suo **bound**, la CPU deve sollevare un'eccezione.

1.7 Segmentazione

1.7.1 Binding

Durante il processo di rilocazione venogno cambiati tutti gli indirizzi del programma per evitare che vadano fuori dallo spazio di indirizzamento previsto. Il **binding** è l'operazione che viene fatta per modificare gli indirizzi. Può essere:

- **Early binding.** Rilocazione degli indirizzi fatta a **compile time**. Il compilatore deve conoscere la posizione di partenza del programma in memoria, ma funziona solo quando il compilatore genera direttamente il codice assoluto (*sistemi embedded, monoprogrammati, ...*).
- **Delayed binding.** La rilocazione degli indirizzi viene fatta durante il trasferimento del programma da disco a memoria (*operazione svolta dal sistema operativo prima dell'introduzione dell'MMU*).
- **Late binding.** La rilocazione degli indirizzi viene fatta immediatamente prima di eseguire l'istruzione corrente, quindi a **runtime**. Per implementare questa tecnica serva l'MMU.

1.7.2 Segmentazione

Con la tecnica base e bound, c'è dello spazio potenzialmente non utilizzato tra lo stack e lo heap. L'idea alla base della **segmentazione** è quella di dividere il programma in **segmenti** che possono essere caricati in porzioni di memoria differenti siccome ad ognuno di essi è associata una coppia base-bound. I segmenti sono inseriti in modo indipendente all'interno della memoria fisica, in questo modo siamo in grado di evitare gli sprechi. Questo risparmio di memoria, tuttavia, complica notevolmente l'MMU, la quale deve gestire più segmenti presenti all'interno della memoria (ogni processo ha tre segmenti). Il meccanismo funziona come segue:

- **Input:** indirizzo logico B (prodotto dal compilatore).

- Individua il segmento s di appartenenza dell'indirizzo B.
- Calcola l'offset k sottraendo all'indirizzo virtuale l'indirizzo di partenza (logico) del segmento ($k = B$ - indirizzo iniziale di s)
- Viene calcolato l'indirizzo fisico sommando k e il base register (Indirizzo fisico = $Base(s) + k$)

Se un processo cerca di produrre un indirizzo illegale, l'hardware rileverà che l'indirizzo è out of bounds, trap nel sistema operativo, il quale terminerà il processo (**segmentation fault**).
L'hardware per conoscere il segmento e l'offset taglia l'address space in segmenti basati sui primi bit dell'indirizzo virtuale (**approccio esplicito**). Nell'**approccio implicito** invece l'hardware determina il segmento in base a come è formato l'indirizzo. Se, ad esempio, l'indirizzo è stato generato dal program counter, appartiene al code segment; se è dello stack o del base pointer, deve appartenere al segmento stack. Ogni altro indirizzo viene interpretato come parte del segmento heap.

1.7.3 Stack

Siccome lo stack cresce al contrario, invece dei soli valori base e bound, l'hardware ha bisogno di sapere in quale direzione cresce il segmento (un bit settato a 1 se il segmento cresce positivamento, 0 negativamente). Il controllo del bound register viene fatto in valore assoluto.

1.7.4 Permessi

Code sharing. Per risparmiare memoria, a volte è utile condividere certi segmenti tra gli address spaces. Per supportare la condivisione abbiamo bisogno di **protection bits** da parte dell'hardware. Vengono aggiunti solamente pochi bit per segmento, a indicare quando un programma può leggerne, scriverne o eseguirne il codice contenuto.

1.7.5 Coarse grained and fine grained

Gli esempi visti fin'ora utilizzavano la tecnica **coarse grained** (poche fette relativamente grandi). Alcuni dei primi sistemi erano più flessibili e permettevano che gli address spaces consistessero in un gran numero di piccoli segmenti, questo concetto era espresso come segmentazione **fine grained**.

Ciò richiede un ulteriore supporto hardware, una **segment table** all'interno della memoria.

1.7.6 Frammentazione

La segmentazione solleva un numero di nuove problematiche:

- Cosa dovrebbe fare il sistema operativo a fronte di un context switch? I segment registers devono essere salvati e ripristinati.
- Come viene gestito lo spazio libero in memoria fisica? Quando un nuovo address space viene creato, il sistema operativo deve essere in grado di trovare lo spazio in memoria fisica per i suoi segmenti.

Il problema generale è che la memoria fisica consuma velocemente piccoli spazi liberi, rendendo difficile l'allocazione di nuovi segmenti o la crescita di quelli già esistenti. Questo problema è noto come **frammentazione esterna**. Si può risolvere con la **deframmentazione**, compattando la memoria fisica e riarrangiando i segmenti esistenti, copiando i dati dei segmenti in una regione contigua di memoria e cambiando il valore dei loro segment registers. Questa operazione è piuttosto complessa e dispendiosa oltre che bloccante. Un approccio più semplice è quello di usare un algoritmo per la gestione della **free-list** che tenta di mantenere un elevato spazio disponibile contiguo in memoria. Purtroppo però la frammentazione esisterà sempre a prescindere da quanto buono sia l'algoritmo per minimizzarla.

1.8 Paginazione

La **paginazione** nasce per gestire in modo ottimale lo spazio libero in memoria e l'address space di un programma. Consiste nel tagliare gli spazi in fette di una certa dimensione. Anzichè dividere l'address space di un processo in segmenti, esso viene diviso in unità di dimensione fissata, ognuna delle quali è chiamata pagina.
Vediamo la memoria fisica come un array di slots di dimensione fissata, chiamati **page frames**. Ogni frame può contenere una singola pagina di memoria virtuale. Ciò porta ad alcuni vantaggi:

- **Flessibilità.** Il sistema sarà in grado di supportare l'astrazione dell'address space efficacemente, a prescidere da come un processo ne fa uso. Non

vogliamo, ad esempio, dover fare assunzioni riguardo la direzione di crescita dello heap e dello stack e come vengono usati.

- **Semplicità** della gestione dello spazio libero. Per esempio, supponiamo che il sistema operativo desideri inserire il nostro addess space da `64B` in memoria fisica. Siccome i programmi sono divisi in pagine di dimensione fissata, il problema della segmentazione viene ridotto di molto visto che, siccome il sistema operativo tiene traccia della free list, gli basta semplicemente prendere il primo frame disponibile e assegnarlo a una pagina.

Per memorizzare dove ogni pagina virtuale dell'address space è posizionata in memoria fisica, il sistema operativo tiene una struttura dati per ciascuno processo nota come **page table**. Il ruolo principale della page table di memorizzare, per ogni pagina virtuale dell'address space, il corrispondente frame fisico.

1.8.1 Address translation

Per tradurre l'indirizzo virtuale generato da un processo, dobbiamo per prima cosa dividerlo in **Virtual Page Number (VPN)** e **offset**. Siccome si conosce la dimensione di ciascuna pagina, si può dividere l'indirizzo virtuale in:

- **VPN:** Bit più significativi che fanno da indice per accedere alla page table del processo per trovare il frame fisico corrispondente (**PFN**).
- **Offset:** Bit che servono per indirizzare la grandezza di una pagina.

A questo punto si traduce l'indirizzo virtuale in fisico sostituendo il **Physical Frame Number (PFN)** al VPN.

1.8.2 Page tables

Le page tables possono essere terribilmente grandi. Per esempio, immaginiamo un address space da `32 bit` con pagine da `4KB`. L'indirizzo virtuale sarà diviso in `20 bit` di VPN e `12 bit` di offset. `20 bit` di VPN implicano 2^{20} possibili traduzioni per ogni processo. Assumendo di aver bisogno di `4B` per **page table entry (PTE)** per mantenere la traduzione fisica più ogni altra informazione utile otteniamo `4MB` di memoria necessari per ogni page table.

Con 100 processi in esecuzione, questo significa che il sistema operativo avrà bisogno di **400MB** di memoria.

Cosa contiene una page table? La page table è sempicemente una struttura dati usata per mappare gli indirizzi virtuali in indirizzi fisici. La forma più semplice è chiamata **page table lineare** che è semplicemente un array. Il sistema operativo indicizza l'array con il VPN e consulta la PTE a quell'indice per trovare il PFN desiderato.
Ogni PTE contiene diversi bit:

- **Valid bit.** Indica quando una particolare traduzione è valida. Per esempio, quando un programma inizia l'esecuzione, avrà code e heap a un'estremità del suo spazio di indirizzamento e lo stack dall'altra. Tutto lo spazio non utilizzato in mezzo sarà marcato come invalido e se il processo tenterà di accedervi, verrà generata una trap al sistema operativo che lo terminerà. È cruciale per supportare un address space sparso.
- **Protection bits.** Indicano quando una pagina può essere letta, scritta o eseguita. Accedere a una pagina in modo non consentito da questi bit genererà una trap nel sistema operativo, il quale terminerà il processo.
- **Present bit.** Indica se la pagina in questione è in memoria fisica o su disco. Consente al sistema operativo di swappare le pagine liberando la memoria fisica.
- **Dirty bit.** Indica se la pagina è stata modificata da quando risiede in memoria.
- **Reference bit.** Viene usato per tenere traccia se una pagina è stata acceduta da quando risiede in memoria.

1.8.3 Quanto è lenta la paginazione?

Per ogni riferimeto a memoria (sia per prelevare un'istruzione che per un load o store esplicito), la paginazione ne necessita uno aggiuntivo per prelevare la traduzione dalla page table. I riferimento a memoria aggiuntivi sono costosi e in questo caso rallenteranno il processo di un fattore pari a due o più.

1.9 Translation Lookaside Buffer

Siccome le informazioni di mappatura risiedono generalmente in memoria fisica, la paginazione richiede un accesso aggiuntivo per ogni indirizzo virtuale generato dal programma. L'obbiettivo è snellire la tecnica introdotta, cercando di **diminuire il numero di accessi a memoria fisica** (alla page table). Viene aggiunta alla MMU una cache hardware delle traduzioni virtual-to-physical più popolari chiamata **translation lookaside buffer o TLB**. Per ogni indirizzo virtuale, l'hardware controlla per prima cosa il TLB per vedere se la traduzione desiderata è presente al suo interno.

```
1         VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2         (Success, TlbEntry) = TLB_Lookup(VPN);
3         if (Success == True){ //TLB HIT
4           if (CanAccess(TlbEntry.ProtectBits == True){
5             Offset = VirtualAddress & OFFSET_MASK;
6             PhysAddr = (TlbEntry.PFN << SHIFT) | Offset;
7             Register = AccessMemory(PhysAddr);
8           }
9           else
10            RaiseException(PROTECTION_FAULT);
11        }
12        else{              //TLB MISS
13          PTEAddr = PTBR + (VPN * sizeof(PTE));
14          PTE = AccessMemory(PTEAddr);
15          if(PTE.Valid == False)
16            RaiseException(SEGMENTATION_FAULT);
17          else if (CanAccess(PTE.ProtectBits) == False)
18            RaiseException(PROTECTION_FAULT);
19          else{
20            TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits);
21            RetryInstruction();
22          }
23
24        }
25
```

L'algoritmo che l'hardware segue funziona in questo modo:

- Estrae il VPN dall'indirizzo virtuale.
- Controlla se il TLB contiene la traduzione per il VPN. Se così fosse, abbiamo un **TLB hit**, la traduzione è cioè contenuta in cache.
- Se la CPU non trova la traduzione nella TLB abbiamo un **TLB miss**.

L'hardware accede alla page table per trovare la traduzione e, assumendo che l'indirizzo virtuale generato dal processo sia valido e accessibile, aggiorna il contenuto del TLB con la nuova entry. Queste operazioni sono parecchio costose.

- Una volta che il TLB è aggiornato, l'hardware riprova l'istruzione, ottenendo un TLB hit.

1.9.1 Performance e località

Il TLB migliora le performance grazie al **principio di località**. Esso si divide in:

- **Spaziale.** Se la CPU sta eseguendo un'istruzione presente in memoria, vuol dire che con molta probabilità le prossime istruzioni da eseguire si troveranno fisicamente nelle vicinanze di quella in corso.
- **Temporale.** Se accedo all'istruzione 100 al tempo t_0, con molta probabilità acederò nuovamente ad essa negli istanti di tempo successivi.

1.9.2 TLB miss

Chi gestisce un TLB miss? Ci sono due possibili risposte:

- **Hardware.** L'HW deve sapere la posizione delle page tables in memoria (attraverso il page table register), oltre al loro formato esatto. In presenza di un miss, l'HW deve accedere alla page table, trovare la PTE corretta, estrarre la traduzione desiderata, aggiornare il TLB con la pagina contenente l'indirizzo fisico ricercato e riprovare l'istruzione.
- **Software (S.O).** Al verificarsi di un TLB miss, l'hardware solleva un eccezione per mettere in pausa il flusso corrente di istruzioni, aumenta i privilegi a livello kernel e salta a un trap handler. Questo trap handler è codice scritto all'interno del sistema operativo, il cui scopo è la gestione esplicita dei TLB misses. Il codice cercherà la traduzione nella page table, userà "speciali" istruzioni privilegiate per aggiornare il TLB e, infinie, eseguirà la **return-from-trap**. A questo punto, l'hardware riproverà l'istruzione (TLB hit).

TLB return from trap In questo caso, quando si torna da una TLB miss-handling trap, l'hardware deve ripristinare l'esecuzione dall'istruzione che aveva causato la trap nel sistema operativo.

Quando il TLB miss-handler è in esecuzione, il sistema operativo deve essere molto attento a non causare una catena infinita di TLB misses. Se ho un miss, viene generata un'eccezione. Bisogna fare un context switch per permettere al S.O. di gestire l'evento. Per mandarlo n esecuzione bisogna mettere l'indirizzo del TLB miss-handler nel PC. Questo indirizzo tuttavia, come tutti gli altri, viene passato all'MMU. Quest'ultima lo cerca nel TLB, ottenenedo un miss. Parte quindi un loop. La soluzione che viene adottata per risolvere questo problema consiste nel tenere il miss handler all'interno del TLB.

1.9.3 TLB - contenuto

Una address-translaion cache tipica potrebbe avere 32, 64 o 128 entries ed essere ciò che viene chiamato **fully associative**. Ciò significa che una traduzione potrebbe essere ovunque nel TLB e l'hardware dovrà cercare in parallelo fino a trovare la traduzione desiderata. Una entry del TLB ha il seguento aspetto: `VPN | PFN | other bits`.
Tra gli other bits generalmente ci sono il **valid bit**, i **protection bits**, ...

1.9.4 TLB - Context Switch

Il TLB contiene traduzioni virtual to physical che sono valide per il processo in esecuzione ma prive di significato per gli altri. Bisogna assicurarsi che quando cambiamo processo, il processo che sta per essere eseguito non usi le traduzioni di quello precedente. Un approccio semplice ma inefficace è fare un **flush** (impostando tutti i valid bit a 0) del TLB a fronte di un context switch. Ogni volta che un processo verrà eseguito, incapperà in TLB misses. Per ridurre questo overhead, alcuni sistemi aggiungono un supporto hardware per abilitare la condivisione del TLB attraberso context switcher. In particolare alcuni sistemi hardware forniscono un campo **Address Space Identifier** (ASID) nel TLB.

2 Concurrency

3 Persistence

4 JOS