Riassunto Operating Systems: Three Easy Pieces (Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau)

A cura di: Stefano di Pasquale e Alessandro Govoni

Capitolo 2 Introduzione OS

L’architettura di Von Neumann ha alla base Fetch, decode e execute nel momento in cui si sta eseguendo un programma.

Il ruolo del OS è rendere il sistema facile da usare, un modo è la virtualizzazione → pertanto si parla di macchina virtuale.

Il SO è responsabile di:

- Esecuzione dei programmi
- Condivisione della memoria
- Interazione con i dispositivi

Per permettere ai programmi e all’utente di accedere alle risorse, il SO rende disponibili delle **System Calls**

C2.1 Virtualizzare la CPU

Il OS virtualizza la CPU, fornendo l’illusione di un’infinità di CPU a disposizione per tutti i processi → per questo più processi possono essere in esecuzione allo stesso momento.

C2.2 Virtualizzazione della memoria

Il modello attuale di memoria fisica è un semplice array di bytes, per leggere / scrivere in memoria occorre specificare un indirizzo.

Si accede continuamente in memoria quando un processo è in esecuzione.

PID = identificatore univoco di un processo in esecuzione.

Ogni processo ha il suo address space privato → il OS mappa la memoria virtuale di ogni processo in una precisa porzione della memoria fisica.

C2.3 Concorrenza

…

C2.4 Persistenza

…

Capitolo 4 Astrazione dei processi

Processo = programma in esecuzione, è astrazione del OS

Il OS virtualizza la CPU → la tecnica usata è il time sharing della CPU → in questo modo gli utenti possono eseguire più processi contemporaneamente (APPARENTEMENTE, ricordiamo che la CPU è una)

Il time sharing rende l'esecuzione inevitabilmente più lenta.

Per implementare la virtualizzazione della CPU occorrono:

- **Meccanismi :** metodi e protocolli a basso livello.
- **Policy**: Algoritmi che permettono al OS di prendere decisioni.

C 4.1 I processi

Machine state = indica cosa un programma può leggere / modificare

Componenti importanti del machine state sono:

- Memoria (e spazio di indirizzamento)

- Registri (IP,PC,SP)
- I/O (liste di file a cui il processo accede)

C4.2 API dei processi

Sono dei metodi per interfacciarsi al OS, le principali API fornite dai moderni OS sono:

- Create → per creare procesi
- Destroy -Z pre distruggere processi
- Wait → far attendere un processo
- Miscellaneous Control → altri controlli sui processi ES. sospensione momentanea di un processo.
- Status → Informazioni di un processo

C 4.3 Creazione dei processi

La creazione di un processo può essere riassunta in alcuni step:

- Load → caricamento codice e dati del programma dl disco all'address space.
- Allocating memory → dopo il carimento del programma, OS alloca memoria necessaria in stack ed heap.
- Other initializzation tasks → Os fa altre inizializzazioni prima di mandare in esecuzione un programma → ES. aprire STDIN STDOUT e STDERR per ogni programma.

Dopo tutte queste operazioni Il OS è pronto a delegare l'esecuzione e altre azioni che lo riguardano alla CPU → che lo eseguirà a partire dalla funzione "Main()".

C4.4 Process States

Il processo può trovarsi in 3 stati:

- Running → Processo in esecuzione
- Ready → processo pronto per essere eseguito
- Blocked → il processo è in una sitauzione di stallo causata da operazione di attesa ES I/O.

C 4.5 Strutture dati

Il OS è un programma che ha delle strutture per tracciare i processi → una di queste è la process list.

OS tiene traccia dello stato dei registri di ogni processo salvandone il contenuto in memoria ogni volta che si passa dall'esecuzione di un programma ad un altro (nel momento in cui il processo viene stoppato)

Quando un altro processo viene mandato in esecuzione si opera il **context switch** → che ripristina lo stato dei registri del nuovo processo attualmente in esecuzione.

Un processo è stato "Initial" quando è in fase di inizializzazione.

Un processo è in stato **Final** il processo è terminato ma deve essere ancora eliminato. (zombie state)

UN metodo per indirizzare le strutture dati usate per la gestione dei processi è il PCB

Capitolo 5 Affrontato in LAB

Capitolo 6 Limited direct execution

OS deve gestire la condivisione della CPU tra i processi mantenendo le performance elevate, efficienza e controllo (impedire ad esempio che un programma si esegua all'infinito o acceda a informazioni riservate).

C 6.1 Limited direct execution

Per risolvere i problemi sopracitati usiamo LDE, che si basa su *eseguire i programmi direttamente su CPU*.

Ovviamente anche questo approccio crea problemi:

- Mantenere il controllo anche quando il programma viene eseguito direttamente.
- Implementare efficacemente time sharing: bloccare un processo per eseguirne un altro.

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

C 6.2 Restricted Operation

Occorre garantire che il processo possa eseguire delle operazioni “riservate” (ES. I/O accedere a risorse di sistema) senza che prenda pieno controllo sul sistema.

Per fare questo si introduce

- user mode → un programma eseguito in user mode ha delle restrizioni (ES. non può fare I/O).
- kernel mode → nella quale viene eseguito il OS → il codice eseguito può “fare ciò che vuole”

Dovendo garantire a tuti i processi di eseguire alcune operazioni normalmente riservate (ES I/O) → il sistema mette a disposizione delle system call

Per eseguire una system call il programma deve eseguire un' istruzione **TRAP,** questa salta nel kernel ed alza i privilegi a kernel mode, viene eseguite le operazioni richieste ed infine ritorna al chiamante con una **return-from-trap** e si torna a user-mode.

Immediatamente dopo la chiamata trap i registri vengono salvati nel kernel-stack; per poi essere ripristinati prima di tornare al programma chiamante.

In fase di boot il kernel inizializza la trap table in cui si specifica il codice da eseguire ad ogni chiamata.

Il OS informa l'hardware delle locazioni dei trap handler.

Ogni system call ha un proprio system call number.

C 6.3 Switching tra processi

Il OS non può effettuare lo switching tra i processi perchè se un proceso è in esecuzione nella CPU, per definizione il OS non è in esecuzione

Approccio cooperativo

Il OS attende che il processo esegua delle system call per riprendere il controllo del flusso di esecuzione.

Utopisco in quanto il OS non ha controllo (ES. un processo può stare per sempre in un loop).

Approccio non cooperativo

Un timer ogni X millisecondi genera un interrupt → il cui interrupt handler restituisce al OS il controllo.

Il timer e l'handler sono configurati in fase di boot.

Salvare e ripristinare il contesto

Quando il OS riprende il controllo, deve decidere se continuare con l'esecuzione del processo blocaato o eseguirne un'altro → tale decisione è presa dallo scheduler.

Se si decide di cambiare processo, il OS esegue del codice a basso livello detto context switch → salva nel kernel stack i valori dei registri del processo bloccato e carica quelli del processo da eseguire.

Si noti che ci sono in realtà due operazioni di salvataggio / ripristino, la prima per passare dal processo A al OS e la seconda per passare da OS al processo B.

OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember addresses of...
	syscall handler
	timer handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A
		...
	timer interrupt	
	save regs(A) → k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) → proc_t(A)		
restore regs(B) ← proc_t(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) ← k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

C 6.4 Problematiche riguardo la concorrenza

Un modo per evitare che timer interrupts blocchino system call

I timer interrupts sono disabilitati durante l'esecuzione di system call.

Capitolo 7 Scheduling

Lo scheduling è una policy ad alto livello del OS che si occupa di gestire i processi.

L'insieme dei processi in "esecuzione" è detta workload.

Allo scopo di comprendere meglio i vari algoritmi di scheduling, partiamo da una serie di assunzioni:

1. Ogni processo rimane in esecuzione per lo stesso tempo.
2. Ogni processo "arriva" allo stesso istante.
3. Una volta iniziata l'esecuzione di un processo, viene eseguito fino al completamento.
4. Tutti i processi usano solamente la CPU (no I/O ecc).
5. Il tempo di esecuzione di ogni processo è noto a priori.

C 7.2 Metrica

Una metrica "è qualcosa che usiamo per misurare un fenomeno", nello studio degli algoritmi di scheduling terremo conto di alcune metriche.

La prima metrica da analizzare è il **turnaround time** (metrica di performance) → è il tempo di completamento dell'esecuzione di un processo. Più formalmente la differenza tra l'istante di completamento e l'istante di "arrivo" del processo.

C 7.3 FIFO

FIFO è una metrica molto semplice → assumendo tre processi (A,B,C) di egual durata e arrivanti allo stesso istante (A un pelo prima di B, B un pelo prima di C) → vengono

eseguiti nell'ordine ABC → al termine del precedente viene eseguito il successivo fino a completamento.

Assumendo che ogni processo duri 10 secondi → il processo A sarà completato in 10 sec. , B in A+10=20 sec., infine C in A+B+10=30 sec. → turnaround time medio = 20 sec.

~~Ogni processo rimane in esecuzione per lo stesso tempo.~~

Uno svantaggio di tale algoritmo è il c.d. convoy effect → se il primo processo è molto "lento" (richiede molto tempo) rallenterà anche i successivi che ne dovranno attendere il completamento.

C 7.4 Shortest Job First (SJF)

SJF manda in esecuzione prima i processi in ordine di durata crescente.

~~Ogni processo "arriva" allo stesso istante.~~

Uno svantaggio è che qualora un processo "breve" (B) arrivi mentre un processo "lungo" (A) è in esecuzione → B dovrà attendere il completamento di A (SJF INEFFICACE).

C 7.5 Shortest Time-to-Completion First (STCF)

~~Una volta iniziata l'esecuzione di un processo, viene eseguito fino al completamento.~~

STCF è in grado di prevedere quale processo (incluso quello corrente) ha il minor tempo di esecuzione rimanente.

Per esempio nel caso stia eseguendo un processo A molto "lungo" e allo stesso istante arrivino 2 processi B e C "brevi" → STCF blocca momentaneamente A per eseguire rispettivamente B e C → riprende poi l'esecuzione di A dal punto in cui lo aveva bloccato.

C 7.6 Altra metrica: Response time

Il response time indica il tempo trascorso tra “l’arrivo” di un processo e la sua prima esecuzione.

Formalmente è la differenza tra l’istante in cui un processo viene eseguito las prima volta e l’istante in cui tale processo è “arrivato”.

Per esempio STCF non ha un buon response time.

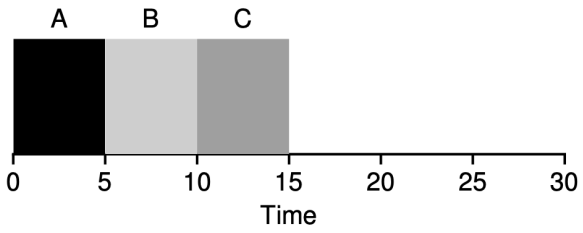

Figure 7.6: **SJF Again (Bad for Response Time)**

C 7.7 Round Robin

Round Robin esegue ciascun processo per una precisa porzione di tempo (time slice) per poi passare al processo successivo

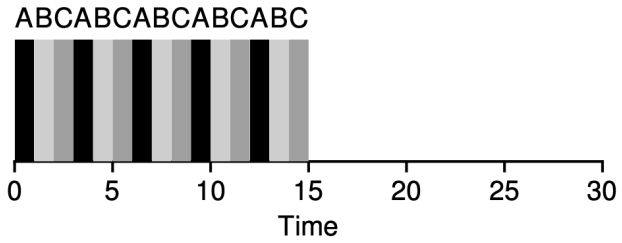

Figure 7.7: **Round Robin (Good For Response Time)**

La durata di ogni time slice deve essere multiplo della durata del timer interrupt.

Round Robin ha un pessimo turnaround time.

Occorre tenere conto che Round Robin esegue una moltitudine di context switch.

C 7.8 I/O

~~Tutti i processi usano solamente la CPU (no I/O ecc).~~

Quando un processo richiede I/O, lo scheduler sa che quel processo per un certo tempo non necessiterà di utilizzare la CPU → pertanto un altro processo può essere mandato in esecuzione.

Viceversa quando un'operazione di I/O è completata, lo scheduler fa ritornare in esecuzione il processo che aveva richiesto l'operazione.

Capitolo 8 Multi-Level Feedback Queue (MLFQ)

~~Il tempo di esecuzione di ogni processo è noto a priori.~~

MLFQ è un algoritmo di scheduling che permette di eseguire i processi assegnando loro una priorità → i processi con priorità maggiore verranno eseguiti prima.

Se due processi pronti ad essere eseguiti hanno la stessa priorità si utilizza l'algoritmo Round Robin.

MLFQ cerca di apprendere il comportamento dei processi per prevederne i comportamenti futuri, per esempio se un processo richiede spesso operazioni di I/O, l'algoritmo ne aumenterà la priorità.

Viceversa se un processo utilizza prolungatamente la CPU, l'algoritmo ne diminuisce la priorità.

C 8.2 Come cambiare priorità

Alcune regole:

- Quando un processo entra nel sistema è posto a massima priorità.
- Se un processo utilizza un intero time slice, la sua priorità è diminuita di uno.
- Se un processo non utilizza un intero time slice, rimane alla stessa priorità.

MLFQ ha 3 principali problemi:

- **Starvation** → Se ci sono molti processi che fanno richieste I/O → potremmo ritrovarci ad eseguire round robin su quei processi non permettendo a processi con priorità più bassa di essere eseguiti.

- **Gaming the system** → Un processo potrebbe prendersi gioco del OS facendo I/O alla fine di ogni suo time slice, in modo tale da mantenere la sua priorità alta.
- **Change Program Behavior** → un programma può cambiare comportamento e diventare molto interattivo ma dato il suo storico il OS lo manterrebbe a bassa priorità.

C 8.3 Boost delle priorità

Per risolvere i problemi di **Starvation e Change Program Behavior,** utilizziamo un priority boost → consiste nel alzare la priorità di ogni processo dopo un certo tempo S.

C 8.4 Miglior gestione del tempo

Se un processo utilizza interamente il suo time slice (non considerando quante volte ha rilasciato la CPU), la sua priorità viene diminuita. (Risolve il problema di **Gaming the system**).

Capitolo 15 Traduzione degli indirizzi di memoria

il OS deve mantenere efficienza e controllo, il controllo implica che un processo non accede ad aree di memoria riservate.

OS deve inoltre garantire flessibilità, lasciando libertà al programma di utilizzare liberamente il suo address space.

Una tecnica generica è **address translation** → hardware modifica ogni accesso in memoria, trasformando l'indirizzo virtuale nel corrispondente indirizzo fisico.

L'hardware non può fare tutto ciò da solo, le operazioni devono essere controllate dal OS.

C 15.1 Alcune assunzioni

Assumiamo che lo spazio di indirizzamento virtuale sia più piccolo della memoria fisica e che ogni spazio di indirizzamento abbia la stessa dimensione.

C 15.3 Rilocazione dinamica

Introduciamo una tecnica denominata base and bunds.

Ci occorrono due registri hardware posizionati nel **memory management unit (MMU)** della CPU:

- Base register
- Bounds

Quando un programma viene eseguito, il OS decide quale porzione di memoria fisica assegnargli → assegna quindi a **Base register** l'indirizzo di memoria iniziale di quella porzione.

Ogni qual volta il programma fa un accesso di memoria, l'indirizzo viene tradotto come segue:

```
physical address = virtual address + base
```

Ad ogni traduzione si verifica che l'indirizzo virtuale risultante non sia maggiore del limite (contenuto nel registro bounds) e che l'indirizzo fisico risultante non sia negativo → in questi casi genera un eccezione → il OS deciderà cosa fare (generalmente termina il processo).

C 15.5 Problemi dei sistemi operativi

Quando viene creato un nuovo processo il OS cerca in una struttura dati (detta free list) per trovare una locazione per il nuovo spazio di indirizzamento.

Quando un processo termina, il OS riottiene tutta la memoria del processo e la rimette nella free list.

Il OS modifica (in kernel mode) il contenuto dei registri Base e Bounds per ogni processo in esecuzione → tali registri vengono salvati ad ogni context switch all'interno del **process control block (PCB).**

Il OS potrebbe decidere di modificare lo spazio di indirizzamento fisico di un processo → per fare ciò blocca l'esecuzione del processo, copia l'intero spazio di indirizzamento dalla locazione corrente alla nuova locazione (copia i dati) e aggiorna il base register presente nel PCB.

OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A:		
allocate entry in process table		
alloc memory for process		
set base/bound registers		
return-from-trap (into A)		
	restore registers of A	
	move to **user mode**	
	jump to A's (initial) PC	
		Process A runs
		Fetch instruction
	translate virtual address	
	perform fetch	
		Execute instruction
	if explicit load/store:	
	ensure address is legal	
	translate virtual address	
	perform load/store	
		(A runs...)
	Timer interrupt	
	move to **kernel mode**	
	jump to handler	
Handle timer		
decide: stop A, run B		
call `switch()` routine		
save regs(A) to proc-struct(A) (including base/bounds)		
restore regs(B) from proc-struct(B) (including base/bounds)		
return-from-trap (into B)		
	restore registers of B	
	move to **user mode**	
	jump to B's PC	
		Process B runs
		Execute bad load
	Load is out-of-bounds;	
	move to **kernel mode**	
	jump to trap handler	
Handle the trap		
decide to kill process B		
deallocate B's memory		
free B's entry in process table		

Figure 15.6: **Limited Direct Execution (Dynamic Relocation) @ Runtime**

Capitolo 16 Segmentazione

Introduciamo la segmentazione per ovviare allo spreco di memoria dovuto all'uso di Base and bounds.

Un segmento è una porzione contigua dell' address space di lunghezza fissata.

Tale divisione permette di posizionare code, stack ed heap in parti diverse della memoria fisica e quindi risparmiare spazio in quanto viene allocata solo la memoria realmente necessaria.

Sono quindi necessarie tra coppie base-bounds per ciascun segmento (code, stack e heap).

Qualora ci si riferisse ad un indirizzo “illecito” si genere un errore di segmentation fault.

C 16.2 Tipi di segmenti

Per conoscere, durante la traduzione, a quale segmento si sta facendo riferimento → i primi 2 bit dell’indirizzo (MSB) lo specificano:

- 00 → code segment
- 01 → heap segment
- 10 → stack segment

I restanti bit dell’indirizzo sono detti **offset.**

Alcuni sistemi mettono code a heap insieme per usare un solo bit.

Approccio implicito: in alcuni sistemi l’hardware determina il segmento sulla base della sua origine. ES. se è generato dal program counter → si tratta del segmento code.

C 16.3 A proposito dello stack

Può essere utilizzato un bit aggiuntivo per specificare se il segmento cresce in direzione positiva o negativa.

C 16.4 Read / write / exec bit

Può essere aggiunta una serie di bit per specificare se un determinato segmento è accessibile in lettura / scrittura / esecuzione.

Se, per esempio, si scrive su un segmento read only l'hardware genererà un' eccezione.

C 16.5 Fine-grained vs. Coarse-grained Segmentation

Coarse-grained → suddivisione dell' address space in poche grosolane parti.

Fine-grained → suddivisione dell' address space in tante piccole parti, quest'ultima richiede il supporto di una segment table salvata in memoria.

C 16.6 OS support

Il OS deve essere in grado di trovare lo spazio libero in memoria per ogni segmento dell' address space generato

Può risultare complesso trovare spazio libero o aumentare la dimensione di un segmento esistente → tale problema è detto **external fragmentation**.

Una soluzione è quella di compattare i segmenti della memoria fisica → tale operazione è computazionalmente dispendiosa.

Un approccio per compattare i segmenti è quello di usare un algoritmo **free list management.**

Capitolo 18 Paging

Un'alternativa alla segmentazione, che tenta di ovviare al problema della **external fragmentation,** è il paging.

Tale approccio consiste nel dividere la memoria in unità di dimensione fissa dette **pagine** → l'insieme delle pagine è un array in cui ogni slot è detto **page frame** → all'interno di ciascun page frame è contenuta una pagina.

Un vantaggio di tale approccio è la flessibilità.

Il OS una lista, detta free list, contenente tutte le pagine libere.

Il OS ha, per ogni processo, una page table ovvero una struttura dati contenente le traduzioni degli indirizzi per ogni pagina virtuale dell' address space.

Per tradurre un indirizzo virtuale generato da un processo, l'indirizzo è diviso in 2 parti:

- Il **virtual page number (VPN)**
- **Offset**

il Physical Frame Number PFN indica un determinato page frame in memoria fisica.

Il PFN si ottiene traducendo il VPN, ad ogni VPN corrisponde un PFN.

L'offset rimane invariato e serve per indicare uno specifico indirizzo (byte) interno alla pagina sia essa virtuale o fisica.

C 18.3 Cosa è contenuto nella page table

Una Page table è composta da una serie di **page table entry,** esistono molti modi per strutturare una page table → il più utilizzato è la **linear page table** (un array).

Il OS cerca all' interno della linear page table all'indice del VPN che si vuole tradurre ed ottiene la corrispondente PFN.

Oltre alla PFN sono presenti una serie di bit di flag:

- Valid bit → ci dice se la traduzione è valida
- Protection bits → read/write/exec
- Present bit → se la pagina è un memoria fisica o nel disco
- Dirty bit → se la pagina è stata modificata da quando è in memoria
- Reference bit → Ci dice se è stata aperta di recente/spesso.

C 18.4

Il processo di traduzione è particolarmente dispendioso e lento → ad ogni accesso in memoria, in realtà se ne effettuano almeno 2 → uno per accedere alla PTE e tradurre l' indirizzo e a seguire l'acceso effettivo.

L’indirizzo della page table è contenuto nel registro CR3 (o Page table base register).

Capitolo 19 TLB

Utilizzare il paging può comportare un peggioramento delle performance, per velocizzare la traduzione degli indirizzi si necessita di un supporto hardware facente parte dell’ MMU, detto **translation-lookaside buffer (TLB).**

Il TLB non è altro che una cache hardware degli indirizzi frequentemente tradotti.

C 19.1 Algoritmo TLB

Assumiamo ci sia una linear page table ed una TLB hardware su cui applichiamo l’algoritmo.

L’ algoritmo compie quanto segue:

Per prima cosa estrae il VPN dall’indirizzo virtuale, controlla se è presente in TLB (quindi se è già stato tradotto in precedenza) → se è presente abbiamo una **TLB hit** → altrimenti una **TLB miss.**

- **TLB hit** → la traduzione è presente in TLB
- **TLB miss** → è necessario eseguire la traduzione ed inserirla in TLB → viene poi richiamato l’algoritmo che produrrà una TLB hit.

C 19.2 Temporal and Spatial Locality

La TLB è avvantaggiata da due concetti:

- Spatial locality → si manifesta quando un programma ripetutamente accede ad indirizzi di memoria vicini tra loro quindi corrisponde quasi sempre ad un TLB hit. Si ha un beneficio perchè nei moderni sistemi le pagine sono di grandi dimensioni.
- Temporal locality → si manifesta quando un programma accede ripetutamente a memoria che è già stata acceduta in precedenza → si ha quasi sempre TLB hit.

C 19.3 Gestione delle TLB miss

La gestione delle TLB miss può essere hardware (CISC) o software (RISC)

- Hardware-managed TLB → necessita di avere page table base register → Oggi poco utilizzata in quanto poco flessibile.
- Software-managed TLB → in caso di TLB miss si genera un'eccezione gestita da una funzione trap → si eseguono le operazioni di traduzione → aggiorna la TLB → si controlla nuovamente la presenza in TLB della traduzione → TLB hit.

Si deve tenere particolare attenzione nel realizzare una Software-managed TLB:

- Il return from trap è diverso dal solito → si deve richiamare l'istruzione che ha causato la trap.
- Occorre evitare loop infiniti di TLB miss → la traduzione dell' indirizzo della trap handler deve essere permanente in TLB oppure in una zona della memoria non soggetta a traduzione.

Software-managed TLB gode di flessibilità e semplicità, il OS può usare qualsiasi tipo di page table senza cambio dell' hardware, il quale deve solo generare un'eccezione in caso di TLB miss.

C 19.4 Contenuto della TLB

Nella TLB per ogni record sono contenuti:

- VPN
- PFN
- altri bit → valid, protection ecc.

C 19.5 Il problema del context switch

Si deve porre particolare attenzione in caso di context switch a non lasciare in TLB traduzioni appartenenti al processo precedente.

Una soluzione è quella di pulire l'intera TLB (Flush) —> operazione molto dispendiosa.

In altri casi si può procedere a porre tutti i valid bit a 0.

Per evitare la flush, si usa TLB condivisa tra i processi → si aggiunge nella TLB un campo identificatore del processo.

Il OS deve specificare all' Hardware l'identificatore del processo attualmente in esecuzione, si utilizza quindi un registro apposito.

C 19.6 Politica di aggiornamento della TLB - 1. Replacement Policy

Quando si inserisce un nuovo record nella TLB occorre ovviamente eliminarne un'altro.

Per scegliere quale record eliminare vi sono alcuni approcci:

- l**east-recently-used LRU** → sfrutta il principio di località elimina il record meno recentemente utilizzato.
- **Random** → utile in casi di cicli su numerose pagine.

Capitolo 20

Se si utilizzano page table troppo grandi si rischia di consumare troppa memoria.

Una soluzione può essere quella di utilizzare pagine più grandi, ciò potrebbe comportare un consumo di spazio interno alla pagina → tale problema è detto i**nternal fragmentation.**

Una soluzione è il c.d. approccio ibrido, che consiste nel combinare il paging e la segmentation al fine di ridurre il consumo eccessivo di memoria delle page table.

Nell'approccio ibrido si ha una page table (e conseguentemente un indirizzo fisico della page table) per ogni segmento (code, stack ed heap)

Ogni segmento avrà la propria coppia di registri base-bounds, all'interno del base register è contenuto l'indirizzo fisico della page table.

In caso di TLB miss, l' hardware, utilizzerà i primi bit per determinare a quale segmento ci si sta riferendo, ricaverà la linear page table relativa a quel segmento, infine traduce l'indirizzo trovando la PTE corrispondente al VPN.

Nel registro buonds è contenuto il numero dell'ultima page table valida.

Tale approccio fa risparmiare spazio rispetto alle linear page table, in quando le pagine non allocate non vengono inserite nella page table.

L’approccio ibrido porta con se gli svantaggi tipici della segmentazione, pertanto si è cercato di implementare meglio page table più piccole.

C 20.3 Multi-level Page table

Un approccio differente è detto **multi-level page table,** consiste nel dividere la page table in unità più piccole di dimensione fissa.

Vine introdotta una nuova struttura (page directory) che contiene gli indirizzi delle sotto-page table valide, ciascun record della page directory è detto page directory entry.

La struttura della page directory entry è simile alla PTE (ha valid bit)

Con tale approccio si allocano solamente le sotto-page table utilizzate (bit valid); inoltre se costruita in modo efficiente ogni sotto-page table può occupare precisamente lo spazio di una pagina.

D’altro canto aumenta la complessità perchè la gestione di questa struttura è più complessa di una struttura lineare, inoltre sono necessari due accessi a memoria per effettuare la traduzione.

Ovviamente è possibile utilizzare strutture con più livelli rispetto a quella presentata in precedenza.

In presenza di TLB, si accede alla Multi-level Page table solo in caso di TLB miss.

Capitolo 21 Meccanismi della memoria fisica

Non sempre è possibile inserire in memoria fisica tutte le pagine, è quindi necessario poter spostare su disco le porzioni dell’address space che risultano poco utilizzate (o non utilizzate di recente).

C 21.1 Swap space

Occorre riservare una porzione del disco, detta **swap space,** dove salvare / da cui caricare le pagine di memoria.

Ovviamente il OS dovrà tenere traccia degli indirizzi del disco (**disk address**) in cui sono salvate le pagine.

C 21.2 Il present bit

In caso di TLB miss, l'hardware controlla nella corrispondente PTE se il **present bit** è 1 → significa che la pagina richiesta è presente in memoria fisica.

Nel caso in cui il **present bit è 0** → la pagina non è in memoria, ma si trova nel disco, si genera un **page fault**.

C 21.3 Il Page Fault

Per gestire il page fault, il OS chiama una funzione detta **page-fault handler.**

Il page fault handler è responsabile di caricare da disco le pagine richieste, l'indirizzo di disco in cui si trova la pagina che si vuole caricare è presente nella PTE corrispondente al posto del PFN.

Una volta caricata la pagina, il present bit viene posto a 1 e inserito nella PTE il corretto PFN. Ora vi sono due scelte possibili:

- Inserire in TLB la traduzione della pagina appena caricata.
- Procedere con un TLB check, il quale produrrà un TLB miss, a cui segue la corretta traduzione.

Mentre si attende l' operazione I/O di lettura da disco, il processo chiamante è in stato blocked, pertanto un altro processo può essere eseguito.

21.4 In caso di memoria piena

Quando la memoria è piena occorre stabilire (attraverso una **page-replacement policy**) quale pagina salvare su disco per lasciare spazio ad altre pagine che devono

essere caricate.

SEGUE NEL CAP 22.

C 21.6 Quando liberare la memoria?

Il OS tende a non riempire mai completamente la memoria fisica, ma ne conserva una porzione libera → per fare ciò si serve di **high watermark** (HW) e **low watermark** (LW).

Quando il OS nota che sono presenti meno pagine rispetto ad LW inizia a liberare memoria (tale processo è detto s**wap deamon o page deamon**) fino a quando vi sono HW (**high watermark)**
 pagine libere.

Per ridurre il numero di accessi al disco, solitamente il OS scrive su disco più pagine in contemporanea.

Capitolo 22 Policy della memoria fisica

Occorre implementare una politica che decida quali pagine rimpiazzare.

C 22.1 Gestione della cache

L’ obbiettivo del OS è minimizzare le occasioni in cui è necessario caricare da disco pagine di memoria (possiamo vedere la memoria fisica come una cache ed il nostro obbiettivo è minimizzare le cache misses).

Possiamo calcolare il average memory access time (AMAT) per ogni programma

```
AMAT = TM +(PMiss ·TD)
TM è il costo di accesso a memoria
TD è il costo dell'accesso a disco
Pmiss è la probabilità di cache miss
```

Nei moderni sistemi il costo di accesso a disco è molto elevato, quindi è necessario ridurre il numero di cache miss al minimo implementando policy migliori.

C 22.2 La politica di rimpiazzo ottimale

Si tratta di un modello ideale non implementabile che si basa sulla capacità di eliminare la pagina che verrà usata nel più remoto futuro.

Ideale poiché il OS non può prevedere il futuro!

Tuttavia è utilizzata come metro di paragone.

Si note che, come in ogni cache si verifica l'effetti **cold-start miss,** i primi accessi ad ogni pagina corrispondono sempre a miss.

C 22.3 Una politica semplice: FIFO

FIFO è una politica molto semplice da implementare ma ha un hit rate molto basso.

C 22.4 Random

Random è un'altra politica relativamente semplice da implementare.

Si basa sul caso quindi è politica volatile.

C 22.5 LRU

LRU tiene conto della frequenza (o dell'attualità) di accesi ad una pagina, la pagina con meno frequenza di accessi o accesso meno recente.

Si basa sul principio di località poiché si basa sul passato per tentare di predire il futuro.

C 22.6 Alcuni esempi

Quando la cache è vuota tutti gli algoritmi performato in ugual modo.

Ovviamente quando la cache contiene tutta la memoria indirizzabile, tutti gli algoritmi performano in ugual modo (100% hit).

Caso 80-20

In cui l'8'% degli accessi è effettuato al 20% delle pagine totali (c.d. pagine hot), il restante 20% degli accessi è effettuato al restante 80% delle pagine.

In questo caso LRU performa meglio di tutti (FIFO e random lavorano abbastanza bene).

Looping sequential

Accediamo in loop 10.000 volte a 50 pagine in sequenza (0-49).

Random risulta il migliore, seppur lontano dall'ottimale.ù

Random risulta essere l'unico algoritmo ad evitare i c.d. **corner case** →

caso in cui l'algoritmo inconsciamente elimina proprio la pagina che gli servirà all'iterazione successiva (cane che si morde la coda).

C 22.7 Implementare algoritmi cronologici

Abbiamo constatato dai confronti effettuati che LRU svolge un lavoro migliore rispetto agli altri (seppur non ottimale).

Gli algoritmi che si basano sulla cronologia sono difficili da implementare in quanto si necessita di modificare una struttura dati ad ogni accesso a memoria, al fine di tenere traccia delle pagine più recente utilizzate, ciò può comportare una riduzione delle performance.

C 22. 8 Il perfetto LRU

Con il supporto dell' hardware si introduce lo **use bit,** uno per ogni pagina. Quando si accede alla pagina lo use bit è posto 1 dall' hardware, il OS si occuperà di azzerare tale bit.

Il OS scorre tutte le pagine alla ricerca di una con use bit = 0 → ipotizzando parta dalla pagina P, se lo use bit di P = 1, lo pone = 0 e salta a P+1.

Nel caso peggiore in cui tutte le pagine hanno use bit = 1, il OS ricomincia il ciclo fino a quando non trova una pagina con use bit = 0.

C 22.9 Pagine Sporche

Il OS fa uso del dirty bit per individuare le pagine che sono state modificate, le quali dovranno essere riscritte su disco prima di essere eliminate dalla memoria.

Tale operazione è compiuta dal **page replace algorithm**.

Come accennato precedentemente il OS è solito scrivere su disco una serie di pagine allo stesso momento, per minimizzarne il costo.

C 22.10 Altre politiche

Per ora abbiamo analizzato solamente casi in cui le pagine vengono caricate solo “su richiesta”; tuttavia esistono algoritmi per prevedere quali pagine verranno richieste in seguito.

Per esempio alcuni algoritmi presumono che se viene caricata una pagina P, probabilmente verrà successivamente richiesta la pagina P+1.

C 22.11 Thrashing

Quando tutta la memoria è piena, il OS può decidere di terminare un processo molto dispendioso in termini di memoria.

Capitolo 26 Concorrenza

Introduciamo ora una nuova astrazione, il **thread.**

Un programma multi-threaded può essere visto come più processi separati, che tuttavia condividono lo stesso address space.

Quando si passa dall’esecuzione di un thread ad un’altro è necessario effettuare un context switch, le informazioni dei thread non in esecuzione sono salvate in thread control blocks.

In un programma multi-thread è presente uno stack per ogni thread, pertanto variabili, parametri, valori di ritorno allocati nello stack sono **thread-local.**

C 26.1 Perché usare i thread?

Le ragioni per cui si utilizza il threading sono principalmente 2:

- Parallelismo
- Evitare che i processi si blocchino in fase di I/O (**overlap** dell' I/O).

C 26.2 Esempi

La funzione pthread_join() attende la terminazione di un thread.

C 26.3 Dati condivisi

Immaginiamo un esempio in cui 2 thread vogliano modificare una variabile condivisa, notiamo che ad ogni esecuzione otteniamo risultati diversi → quindi l'output non è **deterministico**.

C 26.4 Il cuore del problema

In alcuni casi può succedere che un timer interrupt intervenga su un thread prima che quest'ultima abbia completato correttamente le operazioni su una risorsa condivisa, ciò può causare errori o malfunzionamenti.

Siamo di fronte ad una **race condition** quando i risultati di un'operazioni sono dipendenti dall'istante in cui il programma viene eseguito.

Una porzione di codice in cui si può verificare una **race condition** è detta **critical section**.

Ciò di cui abbiamo bisogno per utilizzare correttamente risorse condivise tra più thread sono meccanismi di **mutua esclusione**.

C 26.5 Una soluzione atomica

Una soluzione può essere l'utilizzo di istruzioni assembly che garantiscano la completa esecuzione di quanto desiderato → impercorribile perché aumenterebbe a dismisura il numero di istruzioni macchina.

Una soluzione più valida è quella di utilizzare una serie di **primitive di sincronizzazione** che ci permettano di accedere alle risorse condivise in maniera scura e controllata.

Capitolo 27 Thread API

Nello standard POSIX la funzione pthread_create() prende in ingresso 4 parametri:

- thread → puntatore a struttura di tipo thread.
- attr → definisce gli attributi del thread(es dimensione stack, priorità).
- start routine → puntatore alla funzione da eseguire.
- arg → parametri della funzione eseguita dal thread.

C 27.2 Completamento del thread

Per attendere il completamento di un thread usiamo la funzione pthread_join(), i cui parametri sono:

- il thread di riferimento
- void pointer al valore di ritorno

Si tenga a mente non è possibile far ritornare alla funzione thread una variabile/puntatore a variabile allocata nello stack.

C 27.3 Locks

La libreria POSIX fornisce mutua esclusione di una sezione critica attraverso i **LOCKS**.

Le principali funzione per l'utilizzo dei locks sono:

- int pthread_mutex_lock(pthread_mutex_t *mutex);
- int pthread_mutex_unlock(pthread_mutex_t *mutex);

Usiamo queste funzioni ai margini di una sezione critica per implementare la mutua esclusione.

Quando viene chiamata la funzione lock(), il thread prende il lock → pertanto fino a quando non viene rilasciato l'esecuzione non può essere interrotta.

Quando viene chiamata la funzione unlock(), il lock viene rilasciato.

Occorre inizializzare il lock come segue:

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

oppure in maniera dinamica (a runtime):

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

e per deallocarlo

pthread mutex destroy()

Esistono altre due funzioni:

- trylock() che fallisce se il lock è già preso (non bloccante)
- timedlock() simile a lock() a cui si può specificare un tempo di timout

C 27.4 Variabili condizionali

Abbiamo bisogno di variabili condizionali quando dobbiamo far comunicare due o più thread.

Le due funzioni principali sono:

- wait() → pone il thread chiamante in attesa che un'altro thread lo richiami
- signal() → "sblocca" un altro thread.

Per usare tali funzioni è necessario avere un lock

Capitolo 28 I Lock

C 28.1 Funzionamento lock

Banale. Spiegato precedentemente.

C 28.2

Banale. Spiegato precedentemente.

C 28.4

Per costruire un lock efficiente dobbiamo tenere conto di 4 aspetti:

- Mutua esclusione
- Correttezza → evitare starvation → garantire che un thread “prima o poi” riesca a prendere il lock
- Performance → occorre considerare diversi casi:
  - Singolo thread
  - Più thread su una singola CPU
  - Più thread su più CPU

C 28.5 Gestione degli interrupt

Nel caso di una singola CPU, un metodo per fornire mutua esclusione consiste nel disabilitare gli interrupt.

Gli aspetti negativi sono molteplici:

- Fornire ad un thread la possibilità di compiere l’operazione privilegiata di abilitare/disabilitare gli interrupt non è sicuro.
- Non funzione su più CPU
- Disabilitando gli interrupt per un lungo periodo di tempo rischiamo di perderli.
- Lentezza

C 28.6 Un metodo fallimentare - usare solo load e store

Un metodo per creare un lock potrebbe essere attraverso l’utilizzo di una variabile flag per indicare se un thread qualsiasi è in possesso del lock.

Unlock() → setta flag a 0 → lock disponibile.

Lock() → controlla se flag = 1 si blocca → altrimenti imposta flag=1 → lock preso!

Se la funzione lock flag =1 → ciclicamente controllerà il valore di flag fino a quando non è 0.

Tale approccio ha 2 problemi:

- Correttezza → la funzione lock è se stessa una funzione critica → un thread potrebbe essere interrotto da un timer interrupt prima di settare flag =1 → si rischia che 2 thread abbiano contemporaneamente il lock.
- Performance → ciclare in attesa di flag=0 è perdita di tempo.

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

C 28.7 Costruire spin locks con test and set

Con il supporto dell'hardware creiamo la funzione atomica test and set

```
int TestAndSet(int *old_ptr, int new){
  int old = *old_ptr; // fetch old value at old_ptr
  *old_ptr = new;     // store 'new' into old_ptr
  return old;         // return the old value
}
```

Test and set ricava il valore puntato da *old_ptr (che viene poi ritornato) e lo aggiorna al valore new.

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Per fare si che tale approccio lavori correttamente su una singola CPU

è indispensabile uno scheduler a time slice (es round robin)

C 28.8 Valutazione spin locks

- Correttezza → provvede correttamente alla mutua esclusione
- Parità → un thread potrebbe ciclare per sempre in attesa del lock → starvation
- Performance → occorre considerare due casi

- Singola CPU → spin lock sono molto lento → perdiamo l'intero time slice a tentare di prendere il lock che non si libererà mai.
- Più CPU → spin lock è efficiente.

C 28.9 Confronto e scambia

Un'altra implementazione hardware per fornire mutua esclusione è **compare and swap,** è una istruzione che lavora in maniera simile alla **test and set**

La funzione prende in input 3 valori:

- Indirizzo di memoria da verificare
- Valore atteso
- Nuovo valore

```
int CompareAndSwap( int *ptr , int expected , int new)
{int actual = *ptr ;
i f ( actual == expected )
  *ptr = new;
return actual ;}
```

Il comportamento è sostanzialmente identico a spin lock

C 28.10 Load linked e store conditional

Nelle architetture MIPS sono implementate 2 funzioni:

- load-linked → funzionamento classico di una load
- store-conditional → aggiorna il valore di una cella di memoria solo se la stessa non è stata modificata dopo la load-linked → ritorna 1 in caso di successo, altrimenti 0.

Utilizzando la store-conditional abbiamo la certezza che il valore non sia stato modificato da un altro thread → l'implementazione è corrtta.

Scarse performance per via dei cicli continui e non viene garantita equità.

C 28.11 Fetch and add

Si tratta di una funzione simile a quelle viste in precedenza, in cui il valore contenuto nella cella di memoria viene incrementato.

```
int FetchAndAdd(int *ptr) {
  int old = *ptr;
  *ptr = old + 1;
  return old;
}
```

Con questa primitiva hardware possiamo creare un **ticket lock.**

Un ticket lock prevede un campo condiviso del lock (lock→**turn)** che indica il turno (come fosse un ticket numerico) del prossimo thread che avrà acceso al lock.

Questo approccio assicura che tutti i thread abbiano acceso al lock (fairness).

C 28.12 Troppo ciclare

Come anticipato in precedenza, con una singola CPU, ciclare in attesa che un'altro thread liberi il lock corrisponde ad un totale spreco dell'intero time slice.

C 28.13 Yielding

Un altro approccio è fornire la system call Yield() che un thread può chiamare per rilasciare la CPU, così facendo il thread passa dallo stato running allo stato ready.

Tale approccio risolve il problema dello spin lock (cicli) , tuttavia non è ottimale poiché non fornisce equità → potrebbe verificarsi starvation.

Dal punto di vista delle performance non è ottimale in quanto viene effettuato un context switch per ogni thread in attesa del lock → ciò può essere molto dispendioso.

C 28.14 Utilizo di code

A questi approcci è doveroso aggiungere un supporto da parte del OS, analizziamo l'implementazione di Solaris.

Sono fornite 2 funzioni:

- Park → mette in pausa il thread chiamante
- Unpark → riprende l'esecuzione di un thread

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag  = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock
33                                     // (for next thread!)
34     m->guard = 0;
35 }
```

Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**

Partendo dalle implementazioni precedenti si utilizza una coda in cui ogni thread in attesa di prendere il lock si inserisce, una volta inseritosi in coda il thread chiama park().

Quando un thread vuole rilasciale il lock controlla la coda, se vuota pone flag = 0 altrimenti chiama unpark() sul primo thread in coda.

Si noti che prima di chiamare unpark() il flag rimane a 1 → in questo modo evitiamo che altri thread si inseriscano “saltando la coda” durante il passaggio tra il thread che lascia il lock ed il primo della coda.

Inoltre è necessaria una guardia (uno spin lock) che protegga tutte le operazioni sulla coda e sul flag.

La guardia viene rilasciata dopo la unparck() e prima della park().

Potremmo trovarci in una wakeup/waiting race condition quando tra il rilascio della guardia e la chiamata park() si inserisce un timer interrupt che porta un altro thread a rilasciare il lock (flag = 0 oppure unpark()), il primo thread si troverebbe in una park() senza possibilità di uscita.

Per ovviare a ciò Solaris mette a disposizione una syscall setpark() con cui un thread indica che è “prossimo alla park”.

```
queue_add(m->q, gettid());
setpark(); // new code
m->guard = 0;
```