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

Sito web del libro.

A cura di Stefano di Pasquale e Alessandro Govoni

Capitolo 2 Introduzione sistema operativo

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

Il ruolo del sistema operativo 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**

C 2.1 Virtualizzare la CPU

Il sistema operativo 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.

C 2.2 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 sistema operativo mappa la memoria virtuale di ogni processo in una precisa porzione della memoria fisica.

C 2.3 Concorrenza

Trattato in seguito

C 2.4 Persistenza

Trattato in seguito

Capitolo 4 Astrazione dei processi

Processo = programma in esecuzione, è astrazione del sistema operativo

Il sistema operativo 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 sistema operativo 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)

C 4.2 API dei processi

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

- Create → per creare processi
- Destroy → per 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 dal disco all'address space.
- Allocating memory → dopo il caricamento del programma, sistema operativo alloca memoria necessaria in stack ed heap.
- Other initialization tasks → il sistema operativo fa altre inizializzazioni prima di mandare in esecuzione un programma → ES. aprire STDIN STDOUT e STDERR per ogni programma.

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

C 4.4 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 situazione di stallo causata da operazione di attesa ES I/O.

C 4.5 Strutture dati

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

Il sistema operativo 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 in stato "**Initial**" quando è in fase di inizializzazione.

Un processo è in stato "**Final"** quando 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 6 Limited direct execution

Il sistema operativo deve gestire la condivisione della CPU tra i processi mantenendo performance elevate, efficienza e controllo (deve 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 genera i seguenti 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 sistema operativo → il codice eseguito può “fare ciò che vuole”

Dovendo garantire a tutti 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, vengono 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 sistema operativo informa l’hardware delle locazioni dei trap handler.

Ogni system call ha un proprio system call number.

C 6.3 Switching tra processi

Il sistema operativo non può effettuare lo switching tra i processi perché se un processo è in esecuzione nella CPU, per definizione il sistema operativo non è in esecuzione

Approccio cooperativo

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

Utopistico in quanto il sistema operativo 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 sistema operativo il controllo.

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

Salvare e ripristinare il contesto

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

Se si decide di cambiare processo, il sistema operativo 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 sistema operativo e la seconda per passare da sistema operativo 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

I timer interrupts sono disabilitati durante l'esecuzione di system call per evitare interruzioni non necessarie e garantire che la system call venga completata correttamente.

Capitolo 7 Scheduling

Lo scheduling è una policy ad alto livello del sistema operativo 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 un algoritmo di scheduling 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 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 la prima volta e l'istante in cui tale processo è "arrivato".

Per esempio STCF non ha un buon response time.

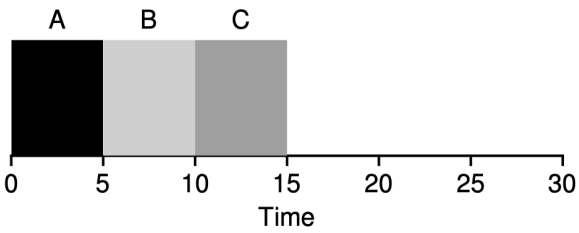


Figure 7.6: **SJF Again (Bad for Response Time)**

C 7.7 Round Robin

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

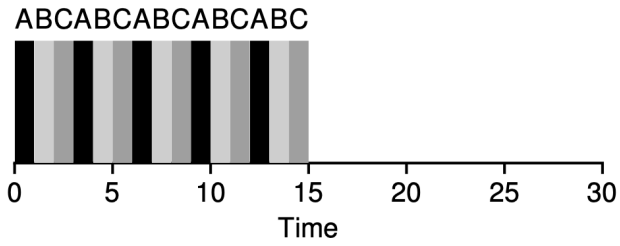


Figure 7.7: **Round Robin (Good For Response Time)**

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

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

Una tecnica generica è **address translation** → l'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 sistema operativo.

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 bounds.

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

- Base register
- Bounds

Quando un programma viene eseguito, il sistema operativo 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 sistema operativo deciderà cosa fare (generalmente termina il processo).

C 15.5 Problemi dei sistemi operativi

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

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

Il sistema operativo 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 sistema operativo 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** (perché assegna una quantità fissa di memoria per ogni indirizzo virtuale, indipendentemente dal fatto che venga utilizzato o meno).

L'indirizzo virtuale di un processo è diviso in segmenti, ciascuno dei quali rappresenta una porzione contigua dell'address space.

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 tre coppie base-bounds per ciascun segmento (code, stack e heap).

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

C 16.2 Tipi di segmenti

Approccio esplicito

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 e heap insieme per usare un solo bit.

Approccio implicito

In alcuni sistemi l’hardware determina il segmento sulla base della sua origine

(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 Segmentation vs. Coarse-grained Segmentation

Coarse-grained → suddivisione dell'address space in poche grossolane 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 sistema operativo 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**.

Essa si verifica quando ci sono blocchi di memoria liberi ma non sufficientemente grandi per soddisfare una richiesta di memoria. Ciò può causare uno spreco di memoria.

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 sistema operativo usa una lista, detta free list, contenente tutte le pagine libere.

Il sistema operativo 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:

- **Virtual Page Number (VPN)**
- **Offset**

La traduzione avviene sostituendo al VPN dell'indirizzo virtuale il suo physical frame number (PFN) presente nella page table, ottenendo l'indirizzo fisico.

Il Physical Frame Number (PFN) indica un determinato page frame in memoria fisica.

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** ognuna delle quali descrive una singola pagina di memoria utilizzata dal processo.

Esistono molti modi per strutturare una page table → il più utilizzato è la **linear page table** (un array).

Il sistema operativo utilizza una linear page table in cui l'indice del VPN è utilizzato come indice per accedere direttamente alla PTE corrispondente nella page table e ottenere il 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 in 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 uno per l'accesso effettivo alla pagina fisica corrispondente.

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 accede ripetutamente ad indirizzi di memoria vicini tra loro quindi corrisponde quasi sempre ad un TLB hit. Si ha un beneficio perché 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
  - farà gestire tutto all'hardware che dovrà conoscere esattamente dove le pagetable sono allocate in memoria tramite il 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 poi si aggiorna la TLB e infine 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 sistema operativo 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.

Per far si che un processo non acceda a zone di memoria non di sua competenza si possono implementare diverse soluzioni:

TLB Flush → cancellazione della TLB dopo un context switch (molto dispendiosa).

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

TLB condivisa tra i processi → si aggiunge nella TLB un Address Space Identifier (ASID) che viene utilizzato per selezionare solo le traduzioni relative al processo in questione, evitando così la flush della TLB per tutti i processi.

Il sistema operativo deve specificare all’Hardware l’Address Space Identifier 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:

- **Least-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 Hybrid page table & Co

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 **internal 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 bounds è contenuto il numero dell'ultima page valida.

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

L'approccio ibrido porta con se gli svantaggi tipici della segmentazione (external fragmentation), 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.

Viene 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 perché 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.

Accessibile grazie al sistema operativo che tiene traccia dei **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 sistema operativo 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.

```
1  PFN = FindFreePhysicalPage()
2  if (PFN == -1)              // no free page found
3      PFN = EvictPage()       // run replacement algorithm
4  DiskRead(PTE.DiskAddr, PFN) // sleep (waiting for I/O)
5  PTE.present = True          // update page table with present
6  PTE.PFN     = PFN           // bit and translation (PFN)
7  RetryInstruction()          // retry instruction
```

Figure 21.3: **Page-Fault Control Flow Algorithm (Software)**

C 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 CAPITOLO 22.

C 21.6 Quando liberare la memoria?

Il sistema operativo 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 sistema operativo nota che la memoria fisica raggiunge **high watermark** (HW), inizia a liberare la memoria per mantenere il livello di utilizzo della memoria fisica entro i limiti stabiliti dal **low watermark** (LW) tale processo è detto **swap deamon**.

Per ridurre il numero di accessi al disco, solitamente il sistema operativo 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'obiettivo del sistema operativo 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 sistema operativo non può prevedere il futuro!

Tuttavia è utilizzata come metro di paragone.

Si noti che, come in ogni cache si verifica l'effetto **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 ha un hit rate molto basso.
Si elimina la prima pagina che è stata inserita in memoria (coda).

C 22.4 Random

Random è un'altra politica relativamente semplice da implementare.

Si elimina la pagina tramite una funzione random

Si basa sul caso quindi non è prevedibile.

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 viene eliminata.

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 performano 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'80% 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.

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 (Algoritmo Clock)

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 sistema operativo si occuperà di azzerare tale bit.

Il sistema operativo 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 sistema operativo ricomincia il ciclo fino a quando non trova una pagina con use bit = 0.

C 22.9 Pagine modificate

Il sistema operativo 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 sistema operativo 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 sistema operativo 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 possono accedere gli stessi dati.

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 (TCBs).

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 → eseguire delle funzioni in contemporanea
- Ottimizzare gli I/O → evitare che i processi si blocchino in fase di I/O (**overlap** I/O) in modo da bloccare solo un thread e continuare l'esecuzione su un altro thread.

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'ultimo 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'operazione sono dipendenti dall'istante in cui il programma viene eseguito.

Questo è dovuto al fatto che più thread stanno cercando di accedere contemporaneamente alle stesse risorse condivise, causando un comportamento imprevedibile o inatteso.

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 sicura 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 funzioni 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 timeout

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() → sveglia il thread in attesa

Per usare tali funzioni è necessario avere un lock.

Capitolo 28 I Lock

C 28.1 Funzionamento lock

Spiegato precedentemente.

C 28.2

Spiegato precedentemente.

C 28.4

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

- Mutua esclusione
- Correttezza → garantire che un thread “prima o poi” riesca a prendere il lock (evitare Starvation)
- 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 funziona 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.

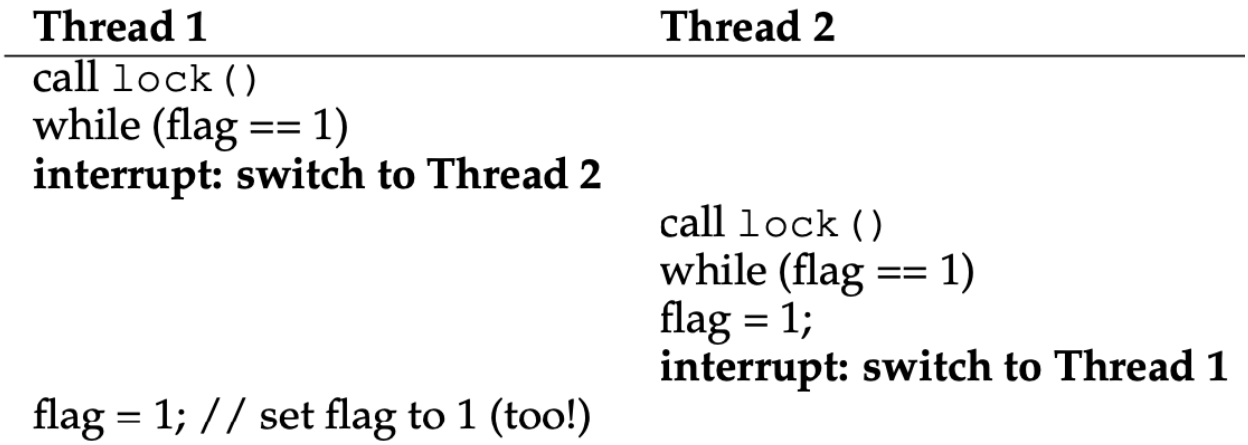
Unlock() → setta flag = 0 → lock disponibile.

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

Se la funzione lock() trova 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 una perdita di tempo.



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 lenti → 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 Compare and Swap

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 che carica in un registro il valore di una variabile;
- store-conditional → aggiorna il valore di una cella di memoria solo se la stessa non è stata modificata dopo la load-linked e 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 corretta.

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 accesso al lock.

Il ticket è assegnato ad ogni thread che desidera ottenere il lock.

La variabile turn viene confrontata con il ticket del thread, se sono uguali il thread otterrà il lock e prima di rilasciarlo aggiorna la variabile turn per permettere al prossimo thread con ticket corrispondente di ottenere il 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 Utilizzo di code

A questi approcci è doveroso aggiungere un supporto da parte del sistema operativo, 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 unpark() 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;
```

Capitolo 29 Strutture dati basate sui lock

Si vuole creare un contatore multi-thread, utile quando si hanno più thread che devono accedere e modificare una stessa risorsa condivisa in modo sicuro e senza causare conflitti tra di loro, esistono vari approcci:

C 29.1 Due approcci semplici

Semplice ma non scalabile

Tale approccio consiste nell’utilizzo di un lock, che ogni thread deve avere prima di incrementare il contatore.

Soluzione efficace ma pessima per quanto riguarda le performance.

Semplice ma scalabile o contatore approssimato

Il counter approssimato fa uso di più counter locali (uno per thread/uno per CPU) ed un counter globale.

Di conseguenza esistono lock per ciascun counter locale ed uno per il counter globale.

Ogni thread/CPU incrementa il proprio counter locale, appena il counter raggiunge una soglia, detta **threshold,** aggiorna il counter globale (ovviamente utilizzando il corrispondente lock) e resetta i counter locali.

Per scegliere il valore di soglia *S* si deve tenere conto che

- Se S è troppo piccolo → si ottiene un comportamento simile alla soluzione precedente (pesante overhead, calo performance)
- Se *S* troppo grande → il counter è poco preciso

C 29.2 Linked-list concorrenti

Esistono vari approcci all'utilizzo di linked-list concorrenti

Linked-list scalabili o hand-over-hand locking

Tale approccio consiste nell' utilizzo di un lock per ogni nodo della lista.

Approccio proibitivo a livello di performance (elevato overhead dato dall'acquisizione/rilascio di ogni lock per scorrere la lista).

C 29.3 Code concorrenti

Esistono 2 lock, uno per la coda e uno per la testa della lista; in questo modo le operazioni **dequeue** e **enqueue** possono essere concorrenti.

C 29.4 Hash table concorrenti

Una hash table può essere implementata utilizzando delle linked-list per la gestione delle collisioni.

Per rendere la hash table concorrente si implementano i lock per ogni lista.

Capitolo 30 Variabili di condizione

In questo capitolo tratteremo come gestire la “comunicazione” tra thread. In alcuni casi un thread è costretto ad attendere il verificarsi di una certa condizione, ritrovandosi in una condizione di **spin wait,** per evitare ciò , si utilizzano le variabili di condizione.

Una **condition variable** è una coda in cui un thread si inserisce (chiamando la funzione wait()) quando è in attesa del verificarsi di una certa condizione.

Quando tale condizione si verifica, altri thread possono “risvegliare” i thread in attesa presenti nella coda (chiamando la funzione signal()).

La funzione wait() deve ricevere come parametro il lock → la stessa, al termine, atomicamente rilascia il lock e mette il thread chiamante in attesa.

Tale approccio potrebbe ricadere in una race condition

```
void thr_exit() {
  done = 1;
  Pthread_cond_signal(&c);
}

void thr_join() {
  if (done == 0)
    Pthread_cond_wait(&c);
  }
```

Se un timer interrupt interviene prima della wait() ed il figlio ora in esecuzione pone done =1 → il thread iniziale rimarrà in attesa per sempre.

C 30.2 Il producer/consumer problem o bounded buffer

Ci si pone nel caso producer/consumer ovvero sono presenti alcuni threads che scrivono su un buffer condiviso (producers) ed altri threads che leggono dati dal buffer (consumers).

Per semplicità considereremo il buffer come un intero.

Esistono 2 funzioni principali:

- Put() → controlla che il buffer sia vuoto e inserisce un valore
- Get() → controlla che il buffer sia pieno e ne ricava il contenuto

Siccome agiscono su un buffer condiviso, le 2 funzioni sopra riportate sono da considerarsi sezioni critiche; di seguito alcune possibili soluzioni per evitare race condition.

Soluzione semplice ma non efficace

Una soluzione può essere di utilizzare **condition variables** (wait() e signal()) sulla condizione di buffer pieno/vuoto.

Tuttavia, si immagini di avere due **consumer thread** (C1 e C2) ed un **producer thread** (P).

Si assuma che C1 viene eseguito per primo, trova il buffer vuoto quindi si mette in attesa.

Di conseguenza viene eseguito P, il quale riempie il buffer e chiama signal() → ora C1 è in stato ready ma non ancora in esecuzione.

Terminata l'esecuzione di P, si immagini che lo scheduler mandi in esecuzione C2, C1 ritorna dalla funzione wait(), rilascia il lock ma trova il buffer vuoto.

Un'altra soluzione semplice ma non efficace - while al posto di if

Una semplice soluzione consiste nel porre la chiamata a wait() all'interno di un ciclo while che verifica la condizione; in questo modo il thread, "al risveglio", controlla la condizione se valida passa oltre, altrimenti richiama wait().

Analogamente si procede a sostituire if con while anche nel producer.

Tuttavia, si immagini di avere due **consumer thread** (C1 e C2) ed un **producer thread** (P).

Si assuma che P sia in esecuzione mentre C1 e C2 siano in attesa, P riempie il buffer e chiama signal() e successivamente si mette in attesa.

Viene eseguito C1 il quale svuota il buffer, chiama signal(), svegliando il primo thread in coda (in questo caso C2), C1 si pone successivamente in attesa.

Viene eseguito C2 il quale dato il buffer vuoto non esce dal while e ritorna in attesa.

Siamo ora in una situazione in cui i 3 thread sono in attesa e non usciranno mai da tale condizione (**deadlock**).

Una soluzione intermedia

Una semplice soluzione consiste nell' utilizzare 2 **condition variables, full (consumer)** ed **empty (producer),** per segnalare lo stato del buffer, in questo modo evitiamo che i consumer "risveglino" i producer e viceversa.

La soluzione generale

Nel caso concreto di più di un buffer, i producers dovranno riempire TUTTI i buffer prima di chiamare signal() e wait().

Viceversa i consumers dovranno svuotare tutti i buffer prima di chiamare signal() e wait().

C 30.3 Covering condition

Nel caso in cui si desideri "risvegliare" tutti i thread in coda, esiste la funzione broadcast().

Ovviamente l'utilizzo di broadcast() comporta un drastico calo delle performance.

Capitolo 31 Semafori

C 31.1 Definizione

Un semaforo è una primitiva di sincronizzazione che permette di sostituire i lock e le condition variables.

Un semaforo è un oggetto che contiene un valore intero che viene modificato attraverso due procedure:

- sem_wait() → decrementa di 1 il valore dell'intero, se il valore attuale è < 0 si mette in attesa, altrimenti passa all'istruzione successiva.
- sem_post() → incrementa di 1 il valore dell'intero, se ci sono uno o più thread in attesa ne risveglia uno.

Prima di utilizzare il semaforo è necessario inizializzarlo tramite la procedura sem_init(), a cui si passa: il puntatore al semaforo, un intero (solitamente 0) e l'intero a cui inizializzarlo.

Si noti che quando l'intero del semaforo è negativo, indica il numero di thread in attesa

C 31.2 Semaforo binario (Lock)

Un semaforo può essere utilizzato come se fosse un lock, per questo utilizzo è necessario inizializzare il semaforo a 1.

Banalmente, quando un thread intende entrare nella sezione critica chiama sem_wait(), uscendo dalla sezione critica chiama sem_post().

Tale implementazione è detta semaforo binario poiché un lock ha solo due stati (preso e libero).

C 31.3 Semafori per l'ordinamento

Possiamo utilizzare i semafori come primitiva di ordinamento (condition variables).

In questo caso si inizializza il semaforo a 0.

Per comprendere il motivo di tale inizializzazione, immaginiamo un esempio in cui un processo padre voglia attendere il completamento di un processo figlio.

Possono verificarsi due casi:

1. Caso in in cui il padre crea il figlio, quest'ultimo NON viene subito eseguito. Il padre chiama sem_wait(), l'intero viene posto a -1 e il padre si mette in attesa. Il figlio,

terminata la sua esecuzione, chiama sem_post(), l'intero è ora 0 perciò il padre viene risvegliato e riprende l'esecuzione.

2. Caso in cui il figlio termina la sua esecuzione prima che il padre possa chiamare sem_wait(). In questo caso il figlio, alla chiamata sem_post() l'intero viene incrementato e raggiunge il valore 1; il padre quando chiamerà sem_wait() decrementerà l'intero (che diventerà 0) quindi non verrà messo in attesa.

C 31.4 Il problema produttore/consumatore

Primo tentativo

Il primo approccio consiste nell'utilizzo di 2 semafori (**empty e full**), full inizializzato a 0 e empty inizializzato a MAX (dimensione massima del buffer)

SI VEDA L' ULTIMO PARAGRAFO DEL CAPITOLO 30.

Immaginiamo di avere 2 thread (un produttore ed un consumatore) su una singola CPU, che il buffer abbia dimensione massima 1 (MAX=1) e che il consumatore venga eseguito per primo.

Il consumatore chiama sem_wait(&full), viene decrementato ed il thread si mette in attesa fino a che il produttore non chiamerà sem_post(&full).

Il produttore chiama sem_wait(&empty), viene decrementato, poiché diventa 0 procede nell'esecuzione, riempie il buffer ed infine chiama sem_post(&full).

Immaginiamo ora che MAX sia > 1 (ad esempio MAX=10) e che vi siano più produttori e più consumatori (race condition).

Assumendo di avere 2 produttori (P1 e P2), P1 inserisce nel buffer un'elemento, ma prima che riesca ad incrementare l'indice del buffer (full) viene interrotto da un timer interrupt. Viene ora eseguito P2 che sovrascrive il valore inserito da P1.

La soluzione: mutua esclusione

La soluzione per ovviare al problema appena citato è la mutua esclusione (lock), ci si trova però in una condizione di deadlock.

Si immagini il caso in cui sono presenti 2 thread (un produttore ed un consumatore), il consumatore viene eseguito per primo, prende il lock e chiama sem_wait(&full). Ora il produttore proverà a prendere il lock, il quale è già preso dal consumatore, si metterà in attesa e nessuno dei due verrà mai risvegliato, siamo quindi in un **deadlock**.

Una soluzione funzionante

Un valida soluzione è quella di mettere il lock solamente a protezione delle operazioni di lettura/scrittura sul buffer (put() e get()).

C 31.5 Reader Writer Locks

Un altro problema comune è quello del Reader&Writer, in cui alcuni thread desiderano scrivere su una lista ed altri leggere (senza modificare) la stessa lista, per risolvere questo problema si implementano i reader-writer lock.

Un solo writer alla volta può accedere alla lista, per farlo chiama rwlock_acquire_writelock() per acquisire il writelock, una volta terminate le operazioni rilascia il writelock chiamando rwlock_release_writelock().

Più reader alla volta possono leggere la lista, per farlo chiamano la funzione rwlock_acquire_readlock() che prende un lock, incrementa un contatore reader e se il thread corrente è il primo reader (readers == 1) prende il writelock chiamando sem_wait(&writelock), alla fine rilascia il lock.

Quando un reader ha terminato le sue operazioni, chiama rwlock_release_readlock(), che prende un lock, decrementa readers, se il thread corrente è l'ultimo reader (readers == 0), rilascia il writelock ed infine rilascia il lock.

Può verificarsi un problema di fairness, nel caso in cui più readers impediscano ad un writer di acquisire il writelock.

In generale si consiglia di utilizzare i reader&writerlock con cautela poiché aumentano di molto l'overhead.

C 31.6 Il problema dei filosofi a cena

Il problema dei filosofi a cena seppur dalla poca rilevanza pratica è un buon modo per riflettere sui problemi di sincronizzazione.

Si immaginino 5 filosofi a cena e 5 posate condivise, ogni filosofo passa il suo tempo a mangiare o pensare, per mangiare ogni filosofo necessità di 2 posate.

Tale problema può essere visto come 5 processi che necessitano ciascuno di 2 dei 5 semafori.

Nel caso in cui ciascun filosofo prenda la forchetta alla sua sinistra, ci si ritroverà in un deadlock in attesa che venga liberata la forchetta di destra.

Soluzione Dijkstra

La soluzione fornita da Dijkstra consiste nel far in modo che uno dei filosofi prenda per prima la posata alla sua destra.

C 31.7 Thread Throttling

Per limitare il numero di thread che compiono determinate operazioni contemporaneamente (esempio una sezione di codice memory-intensive), si usa un semaforo a protezione della sezione di codice interessata.

Il semaforo viene inizializzato al numero massimo di thread contemporaneamente all'interno della sezione.

C 31.8 Come implementare i semafori

Un'implementazione dei semafori diversa da quella di Dijkstra è detta, Zemaphores.

Tale implementazione (usata su Linux) non tiene traccia del numero di thread in attesa, in questo modo l'intero contenuto nel semaforo non sarà mai negativo.

Capitolo 36 I/O Devices

In questo capitolo introdurremo come il sistema operativo interagisce con gli I/O devices.

C 36.1 Architettura del sistema

I dispositivi che necessitano di alte prestazioni vengono connessi “più vicini” alla CPU, solitamente attraverso connessioni PCIe detti **General I/O Bus**.

Gli altri dispositivi, che richiedono minori prestazioni sono “più distanti” dalla CPU e collegati con **Peripheral I/O Bus** ad esempio SATA e USB.

Solitamente la CPU interagisce con un chip I/O (ad esempio Intel Direct Media Interface) a cui sono collegate le periferiche che richiedono minori prestazioni; mentre le periferiche che richiedono maggiori prestazioni sono collegate direttamente alla CPU (es. scheda grafica).

C 36.2 Device canonico

Un I/O device è composto da due parti principali:

- **Interfaccia** → insieme di registri che vengono resi disponibili al sistema per interagire con il device
- **Struttura interna** → è la struttura propria del dispositivo, che permette l’astrazione necessaria per il funzionamento.

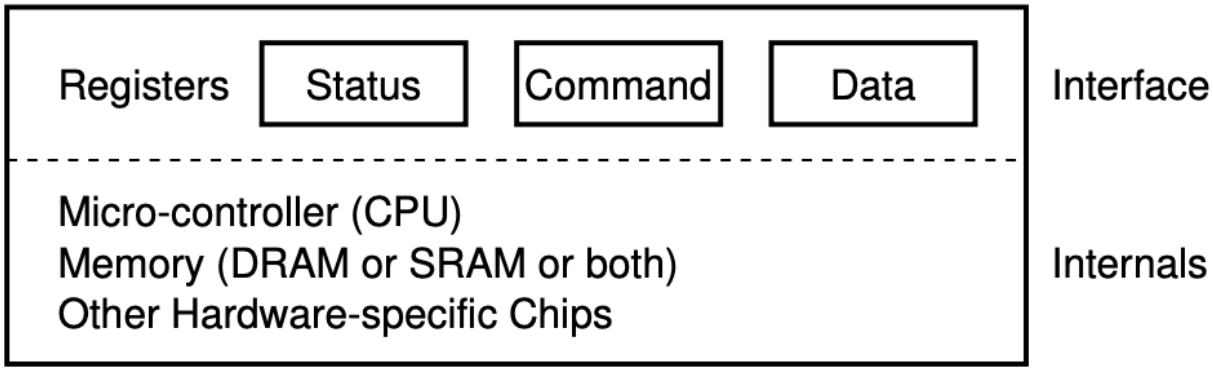


Figure 36.3: **A Canonical Device**

C 36.3 Protocollo Canonico

Nel nostro esempio (Figure 36.3) le interfacce corrispondono a 3 registri:

- **Status register** → Contiene lo stato attuale della periferica
- **Command register** → Usato per indicare al device le operazioni da eseguire
- **Data register** → Usato per leggere/scrivere dati sulla periferica

Di seguito un esempio di interazione tra sistema operativo e il device.

```
While (STATUS == BUSY)
       ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
      (starts the device and executes the command)
While (STATUS == BUSY)
       ; // wait until device is done with your request
```

Nella prima istruzione si attende che il device sia libero e pronto per l'interazione, tale approccio è detto polling.

Nella seconda riga il sistema operativo invia dati al registro **data,** tale approccio in cui la CPU è direttamente interessata nel trasferimento dei dati è detto **programmed I/O**.

Nella terza riga il sistema operativo scrive il comando nel command register, così facendo indica al device che può iniziare a lavorare sui dati.

Infine il sistema operativo rimane in attesa che il device finisca l'esecuzione, utilizzando il polling.

Questo approccio ha alcuni difetti, principalmente il polling tiene occupata la CPU per tutto il tempo di attesa del device.

C 36.4 Diminuzione del CPU overhead utilizzando gli interrupt

Un approccio alternativo al polling consiste nel mettere in attesa il processo quando richiede I/O, il device genererà un interrupt, al termine delle operazioni, che verrà gestito attraverso un **interrupt handler.**

L’interrupt handler corrisponde ad una serie di righe di codice del sistema operativo necessarie per completare la richiesta (lettura dati e codici errore) e risvegliare il processo in attesa.

L’utilizzo degli interrupt permette l’**overlap** (sovrapposizione) tra computazione della CPU ed operazioni di I/O.

Tuttavia l’utilizzo degli interrupt non è sempre la migliore soluzione come nel caso di un operazione molto breve.

Se la velocità del device non è nota a priori, si utilizza il modello **ibrido,** il quale effettua il polling per un breve periodo di tempo, se il device non risponde immediatamente verrà gestito tramite interrupt.

Un altro svantaggio dell’utilizzo degli interrupt si verifica, per esempio, nel caso in cui si verificano un numero elevato di interrupt in un tempo ravvicinato, in questo caso siamo costretti a dover gestire gli interrupt bloccando di fatto la normale computazione.

Una soluzione al problema appena presentato è detta **coalescing,** consiste nel attendere altre richieste prima di inviare l’interrupt.

C 36.5 Utilizzo del DMA

Un altro problema del programmed I/O si ha quando grandi quantità di dati vengono spostati dal device I/O (o viceversa), l’operazione di trasferimento tiene occupata la CPU.

Una soluzione è il **Direct Memory Access,** un device interno all’architettura (un chip) che si occupa di gestire il trasferimento tra i device e la memoria principale senza l’intervento della CPU.

In caso di necessità il sistema operativo indica al DMA dove si trovano i dati da copiare e a quale device inviarli, al termine della copia il DMA genera un interrupt.

C 36.6 Metodi di interazione con i devices

Nel tempo si sono sviluppati due metodi di comunicazione con i devices

- **I/O instructions** → istruzioni macchina specifiche per comunicare con i device, tali operazioni sono da intendersi privilegiate.
- **Memory mapped I/O** → I registri di ogni device vengono trattati come se fossero celle di memoria.

Entrambi non sono caratterizzati da particolari vantaggi.

C 36.7 I driver

Il sistema operativo necessita di conoscere il funzionamento di un device, il codice contenente le specifiche di comunicazione con un device è detto **Device driver.**

Esaminiamo il file system di Linux.

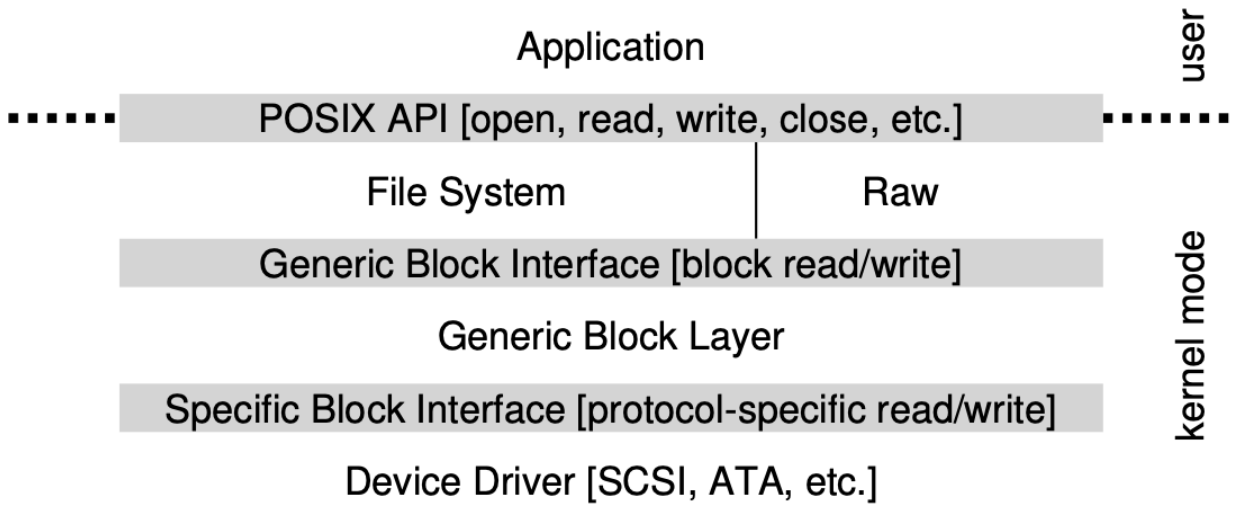


Figure 36.4: **The File System Stack**

Un file system è ignaro delle caratteristiche del disco che sta utilizzando, semplicemente emette richieste di lettura/scrittura al **Generic block layer** che le instrada al **Device driver** appropriato, il quale gestisce i dettagli della richiesta.

Il **generic block layer** è una componente del kernel che si occupa di gestire le richieste di tutte le periferiche nel sistema.

Inoltre è presente una **raw interface** che permette ad applicazioni specifiche di leggere direttamente i blocchi senza l'uso dell'astrazione dei file.

Oggi i driver corrispondono a circa il 70% del codice del sistema operativo, essendo solitamente programmati da programmatori non specializzati in kernel sono più vulnerabili.

C 36.8 IDE Disk driver

L'interfaccia di un disco IDE è composta da 4 tipi di registro

- Control register
- Command block register
- Status register
- Error register

Questi registri sono disponibili per lettura/scrittura (operazioni **IN** e **OUT** su x86) come indirizzi di memoria di I/O.

Il processo di interazione con il device è il seguente, assumendo che il device sia già stato inizializzato.

- SO si mette in attesta che il disco sia pronto: si legge il registro di stato finché da BUSY passa a READY.
- SO scrive i parametri nel command register: viene scritto il sector count, il logical block address (LBA) dei settori a cui si deve accedere e il numero di drive.
- SO fa iniziare I/O scrivendo il comando READ o WRITE nel command register.
- Avviene il trasferimento dati (per le scritture): si aspetta finché il dispositivo passa dallo stato READY a DRQ (Drive ReQuest for data) per scrivere i dati nel registro data port.
- Handle interrupts: nel caso semplice viene generato un interrupt ad ogni settore trasferito, più complesso è l'approccio che permette batching e un interrupt finale al termine del trasferimento.
- Gestione degli errori: per ogni operazione si legge il registro di stato, se il bit di Errore è a 1 si legge il registro errori per i dettagli.

Il protocollo lavora attraverso 4 funzioni principali

- **ide_rw()** → mette in coda una richiesta (se altre sono presenti), altrimenti chiama direttamente **ide_start_request();** in entrambi i casi il processo chiamante viene messo in attesa.
- **ide_start_request()** → usata per inviare richieste di lettura/scrittura al disco, usa la funzione ide_wait_ready() per assicurarsi che il device sia libero prima di inviare la richiesta
- **ide_intr()** → invocata quando viene generato un interrupt, legge i dati dal device e sveglia il processo che è in attesa di I/O, nel caso in cui ci siano altre richieste di I/O le invia con **ide_start_request()**.

Capitolo 37 Hard Disk Drives

C 37.1 L'interfaccia

L'unità disco consiste in un grande numero di settori (blocchi di 512 byte) ciascuno dei quali può essere letto e scritto. Possiamo numerare i settori da 0 a N-1 (per un HDD di N settori), perciò possiamo vedere il disco come una array di settori, tale array è detto **address space** del drive.

Molti file system leggono o scrivono 4KB alla volta, ma i produttori garantiscono solamente una scrittura atomica di 512 byte.

Possiamo assumere che accedere a due blocchi contigui (sequential read/write) sia più veloce rispetto ad un pattern di acceso casuale.

C 37.2 Geometria di base

Il disco si compone di :

- Un **platter,** una superficie circolare su cui i dati sono scritti persistentemente inducendo cambiamenti magnetici, ogni platter ha 2 lati ognuno di questi è detto **Surface.** Un HDD può avere più di un platter.
- Un **mandrino** (o **spindle**) connesso ad un motore, che fa girare i platter ad una velocità costante. Le rotazioni sono misurate in **Rotations Per Minute** (**RPM**).

- **Tracce** concentriche su ogni superficie del platter.
- Una **testina** (**disk head**) presente per ogni superficie.
- Ogni testina è connessa ad un **braccio** (**disk arm**) che permette di muovere la testina sulla superficie al fine di posizionarla sulla corretta traccia.

C 37.3 Un semplice HDD

Assumiamo di avere un semplice disco con una sola traccia.

Tale traccia contiene 12 settori da 512 bytes ciascuno.

Rotation delay

Quando una richiesta di lettura/scrittura arriva al disco, è necessario attendere che, grazie alla rotazione, il settore richiesto si posizioni sotto alla testina.

Il tempo necessario per compiere tale rotazione è detto **rotation delay**.

il tempo di un'intera rotazione si indica con R.

Tracce multiple: seek time

Assumiamo ora di avere una superficie con 3 tracce.

Quando una richiesta di lettura/scrittura arriva al disco, è necessario muovere il braccio sulla corretta traccia, tale processo è detto **seek**.

Una volta posizionata la testina sulla traccia corretta è necessario attendere che il settore richiesto si posizioni sotto di essa.

Altri dettagli

Solitamente i settori vengono posizionati in modo tale da rendere efficiente una lettura/scrittura sequenziale anche nel caso in cui sia necessario cambiare traccia.

Per ovvie implicazioni geometriche le tracce più esterne avranno più settori rispetto alle tracce interne.

I moderni HDD hanno una cache usata per memorizzare dati da scrivere o appena letti dal disco.

Per esempio, quando il disco riceve una richiesta di lettura, legge e mette in cache tutta la traccia, in modo da averla disponibile in caso di richieste successive.

Quando il disco deve servire una richiesta di scrittura ha 2 possibili scelte:

- Segnalare l'avvenuta scrittura appena il dato viene inserito in cache (**write back caching**).
- Segnalare l'avvenuta scrittura solo quando il dato è stato effettivamente scritto (**write through**).

Ciascuno dei due metodi ha i propri pregi e difetti che verranno approfonditi nei capitoli successivi.

Tempi di I/O: facciamo i conti

Possiamo considerare il tempo di I/O come la somma di 3 componenti.

$$Ti/o = Tseek + Trotation + Ttransfer$$

Invece il rapporto di I/O (I/O ratio, R I/O) solitamente utilizzato per confronti fra dischi si calcola come segue

$$RI/O = (Sizetransfer)/(TI/O)$$

Solitamente si analizzano due carichi di lavoro

- **Random workload** → lettura di settori randomici
- **Sequential workload** → lettura di numerosi settori contigui

In questo capitolo si commentano i passaggi per calcolare R I/O di due dischi al fine di confrontarne le prestazioni.

In questo riassunto si è deciso di omettere il procedimento, di seguito le caratteristiche dei dischi ed i risultati ottenuti.

	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

Figure 37.5: **Disk Drive Specs: SCSI Versus SATA**

	Cheetah	Barracuda
$R_{I/O}$ Random	0.66 MB/s	0.31 MB/s
$R_{I/O}$ Sequential	125 MB/s	105 MB/s

C 37.5 Disk Scheduling

Il **disk scheduler** si occupa di organizzare e gestire le richieste di I/O, a differenza di quanto avviene con i processi, il disk scheduler può facilmente prevedere il tempo necessario a soddisfare una richiesta di I/O.

Il disk scheduler cercherà di seguire il principio **SJF (Shortest job first).**

SSTF: Shortest Seek Time First

Tale algoritmo ordina la coda delle richieste in base alla traccia per minimizzare gli spostamenti del braccio.

Tuttavia si presenta un problema di **Starvation,** si immagini un flusso continuo di richieste per la traccia su cui la testa è attualmente posizionata, le richieste per altre tracce non verrebbero mai servite.

Elevator (a.k.a. SCAN or C-SCAN)

Per ovviare al problema di Starvation, è stato ideato l'algoritmo **SCAN (**detto algoritmo dell'ascensore**).**

SCAN fa muovere la testina dalle tracce più interne a quelle più esterne e viceversa (tale movimento è detto **sweep**), perciò se arriva una richiesta per una traccia che è già stata servita nello sweep corrente, si deve attendere lo sweep successivo.

Esistono una serie di varianti dell'algoritmo SCAN, tra cui

- **F-SCAN** → blocca la coda mentre sta effettuando uno sweep (le richieste che arrivano dopo l'avvio dello sweep vengono elaborate nella scansione successiva), in questo modo evita la Starvation di richieste "lontane".

- **C-SCAN** → lo sweep avviene solo dall'esterno verso l'interno (non appena la testina raggiunge l'altra estremità, ritorna immediatamente all'inizio del disco senza soddisfare alcuna richiesta durante il ritorno), garantisce così maggiore fairness nei confronti delle tracce ai margini.

Tuttavia SCAN non si avvicina come vorremmo al **SJF** e ignora la rotazione.

SPTF: Shortest Positioning Time First

Tale algoritmo tiene conto sia del seek time sia del rotation time, ordina la coda in base al tempo di posizionamento minore.

Tuttavia è un algoritmo difficile da implementare su un sistema operativo, il quale per sua natura non conosce i dettagli del disco.

Altri problemi dello scheduling

Come accennato nel paragrafo precedente, il sistema operativo non conosce i dettagli del disco; pertanto nei moderni sistemi lo scheduling avviene internamente al disco.

Un altro importante compito dello scheduler del disco è detto **I/O merging,** consiste nell'unire richieste a blocchi contigui in una singola richiesta di lettura/scrittura di più

blocchi.

Tale operazione è solitamente fatta dal sistema operativo ed ha lo scopo di ridurre il numero di richieste a disco (minore overhead).

Vi sono due approcci nell'invio di richieste I/O al disco

- **work-conserving** → consiste nell'inviare immediatamente la richiesta al disco
- **non-work-conserving** → consiste nell'attendere qualche istante prima di inviare la richiesta al disco poiché potrebbe arrivare una nuova e "migliore" richiesta.

Numerose ricerche hanno dimostrato che l'approccio **non-work-conserving** è migliore**.**

Capitolo 38 RAID

In questo capitolo introdurremmo il **RAID** (**Redundant Array of Inexpensive Disks**) una tecnica per usare più dischi per creare un sistema di dischi più veloce, più grande e più affidabile.

Dall'esterno un RAID è come un comune disco, può essere visto come una serie di blocchi che possono essere letti e scritti.

Internamente si compone di più dischi, memoria (sia volatile che non) ed uno o più processori.

RAID offre una serie di vantaggi :

- Performance → più dischi in parallelo riducono i tempi di I/O
- Capacità
- Affidabilità → la ridondanza permette al sistema di funzionare normalmente anche a seguito della perdita di un disco.

Il RAID fornisce tali vantaggi **trasparentemente** al sistema operativo, al quale si mostra come un unico grande disco; ciò permette di sostituire un semplice disco con un RAID senza modificare una singola riga di codice.

C 38.1 Interfaccia e interno del RAID

Internamente un RAID è composto da una serie di microcontrollori che eseguono un firmware per accedere ai dischi, memorie volatili DRAM usate come cache e memoria non volatile.

Ad alto livello è del tutto simile ad un computer che esegue software specializzati per operare il RAID.

C 38.2 Modello di guasto

Il modello di guasto che assumiamo è particolarmente semplice, è detto **fail-stop**; prevede che il disco possa essere solamente in 2 stati: **working** o **failed (**permanentemente perso**)**.

Assumiamo anche che il fallimento di un disco viene riconosciuto.

C 38.3 Come valutare un RAID

Esistono differenti approcci per realizzare un RAID, valuteremo ciascun design su tre aspetti

- Capacità
- Affidabilità
- Performance

C 38.4 RAID livello 0 - Striping

Di fatto non si tratta di un RAID, non vi è nessuna ridondanza.

Prevede di distribuire i blocchi dell’array sui vari dischi, tale approccio ha il maggior parallelismo per richieste a blocchi contigui.

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

I blocchi su una stessa riga sono detti **stripe**.

I blocchi potrebbero essere disposti in modo diverso, in modo da avere più blocchi contigui nello stesso disco; tale numero è detto **chunk size**

Disk 0	Disk 1	Disk 2	Disk 3	
0	2	4	6	chunk size:
1	3	5	7	2 blocks
8	10	12	14	
9	11	13	15	

Chunk size

Un chunk size piccolo implica sia maggiore parallelismo che tempo di posizionamento.

D’altra parte, una chunk size grande riduce sia il parallelismo che tempo di posizionamento.

Nel resto del capitolo considereremo una chunk size di 4KB.

Valutazione performance di RAID 0

Per valutare le performance di RAID consideriamo due metriche

- **Single request latency**
- **Steady state throughput** → totale larghezza di banda di parecchie richieste concorrenziali.

Siccome RAID è usato in ambienti ad elevate prestazioni, steady state è ovviamente il parametro più importante da tenere in considerazione.

Come nel capitolo precedente, il calcolo delle performance viene effettuato in due contesti

- Random workloads
- Sequential workloads

In conclusione, in RAID 0 la latenza di una singola richiesta è equiparabile a quella di un singolo disco;

Throughput :

- nel caso di accessi random è $= N * R$ MB/s ;
- nel caso di accessi sequenziali è $= N * S$ MB/s .

C 38.5 RAID livello 1 - Mirroring

Con RAID 1 i dati vengono scritti su più dischi contemporaneamente, creando una copia esatta dei dati su ognuno, così facendo, in caso di disk failure, i dati sarebbero ancora disponibili sul disco di backup.

Assumiamo che per ogni blocco esistano 2 copie.

In caso di lettura è possibile scegliere quale copia leggere; invece in caso di scrittura è necessario scrivere entrambe le copie.

Analisi di RAID 1

RAID 1 è molto costoso, solo metà della capacità totale è effettivamente utilizzabile a causa dell'utilizzo di due dischi uguali (nel nostro esempio).

Dal punto di vista dell’affidabilità è possibile sopperire alla rottura di almeno un disco, ma nella migliore delle ipotesi fino a N/2 rotture.

La latenza di una singola richiesta di lettura è equivalente a quella di un singolo disco; nel caso di una singola scrittura, vengono effettuate due scritture in parallelo, il tempo di completamento è pari a quello dell’ operazione con maggiore seek time. Il costo è comunque leggermente maggiore di una lettura su singolo disco.

Analizzando steady-state throughput, otteniamo che nel caso di scrittura la migliore larghezza di banda è $N/2 * S$.

A differenza di quanto si possa pensare, anche una richiesta di lettura di blocchi contigui ha una larghezza di banda $N/2 * S$ MB/s. Per capire il motivo si pensi al fatto che un singolo disco (nel peggiore di casi) riceverà richieste di lettura non contigue, ciò è dovuto alla suddivisione dei blocchi tra i dischi.

Nel caso di letture random, RAID 1 ha buone prestazioni, si ottiene la massima larghezza di banda $N * R$ MB/s , mentre scritture random $N/2 * R$ MB/s.

C 38.6 RAID livello 4

RAID 4 per avere ridondanza utilizza il concetto di parità per ovviare alla ridotta capacità di RAID 1

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Come è possibile evincere dall'immagine per ogni riga è stato inserito un blocco di parità, per calcolare la parità si fa uso dell' operatore logico XOR.

XOR su una serie di bit ritorna 0 se è presente un numero pari di 1, ritorna 1 se è presente un numero dispari di 1.

Su ogni riga deve esserci (includendo i bit di parità) un numero pari di 1 → questa è la invariante che RAID 4 deve mantenere.

Per ricostruire i bit di un blocco andato perduto, occorre fare XOR tra tutti gli altri blocchi della fila, includendo il blocco di parità.

Si specifica che XOR viene effettuato tra bit dello stesso “peso” (nella stessa posizione) di ogni blocco.

RAID 4 Analisi

La capacità di RAID 4 è $(N-1)*B$, dal punto di vista dell’affidabilità RAID 4 tollera la rottura di un singolo disco.

Per quanto riguarda le performance, nel caso di una lettura sequenziale possiamo utilizzare N-1 dischi, per una larghezza di banda di $(N-1)*S$ MB/s.

Nel caso di una scrittura sequenziale, il caso migliore consiste nella scrittura di una intera riga, larghezza di banda $(N-1)*S$ MB/s.

Analizzando letture random otteniamo velocità $(N-1)*R$ MB/s.

Il caso più interessante è quello di scritture random, una volta scritto il nuovo dato è necessario aggiornare il blocco di parità, questa operazione può essere compiuta in 2 modi

- **additive parity** → leggere tutti gli altri blocchi, fare XOR tra blocchi letti e blocco appena inserito, richiede N letture
- **subtractive parity** → legge il “vecchio” valore del blocco (C_old) e “vecchio” valore di parità (P_old), se C_old e C_new sono uguali allora P_new = P_old; altrimenti P_new = opposto di P_old.

$$Pnew = (Cold \oplus Cnew) \oplus Pold$$

⊕=XOR

Ovviamente l’operazione è ripetuta per ogni bit del blocco.

Si immagini ora che due richieste di scrittura per due blocchi random arrivino quasi allo stesso istante, è facile intuire che la lettura del disco di parità necessaria per il subtractive method rappresenta un **collo di bottiglia** (le letture non possono avvenire in parallelo).

Tale problema comporta una pessima velocità di $R/2$ MB/s.

La latenza di una singola lettura è equivalente al caso di un singolo disco. Una richiesta di scrittura invece comporta 2 letture e 2 scritture (in entrambi i casi possono avvenire in parallelo), la latenza è quindi doppia rispetto al singolo disco.

Small write Problem

(In parole semplici)

Il problema delle small write nei sistema RAID-4 basati sulla parità consiste nella lentezza del sistema nel gestire piccoli write su più dischi contemporaneamente.

Questo è dovuto al fatto che, ogni volta che viene effettuata una scrittura, il sistema deve calcolare la parità e scrivere il blocco di dati su tutti i dischi, anche se la quantità di dati effettivamente scritta è molto piccola. Questo può rallentare notevolmente il sistema e aumentare la latenza delle operazioni di scrittura.

Le conseguenze di questo problema sono una riduzione della prestazione del sistema, un aumento della latenza delle operazioni di scrittura e un'usura prematura dei dischi a causa della maggiore attività di scrittura. Questi fattori possono compromettere la disponibilità e la integrità dei dati nel sistema RAID-4 e ridurre la sua efficienza nel proteggere i dati in caso di guasti dei dischi.

C 38.7 RAID livello 5

RAID 5 lavora in maniera simile a RAID 4 con l'unica differenza che i blocchi di parità sono inseriti a rotazione in tutti i dischi.

Tale soluzione risolve (almeno in parte) il problema del collo di bottiglia citato precedentemente.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

RAID 5 Analisi

I tempi di latenza per una singola richiesta sono del tutto simili a quelli di RAID 4.

Le performance di scrittura sono decisamente migliori rispetto a quelle di RAID 4, $N/4 * R$ MB/s.

Viste le sue performance, nel tempo RAID 5 ha quasi completamente rimpiazzato RAID 4.

C 38.8 Sommario della comparazione tra RAID

Si tenga conto che nelle analisi precedenti sono state inserite delle dovute semplificazioni.

La tabella seguente riassume le essenziali differenze tra RAID.

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	$N \cdot B$	$(N \cdot B)/2$	$(N-1) \cdot B$	$(N-1) \cdot B$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S$[1]	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S$[1]	$(N-1) \cdot S$	$(N-1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} R$
Latency				
Read	T	T	T	T
Write	T	T	$2T$	$2T$

Capitolo 39 Files & Directories

In questo capitolo si tratteranno le interfacce utilizzate per interagire con un file system UNIX

C 39.1 Files e Directories

Un file può essere visto come un array di bytes, ciascuno dei quali può essere scritto o letto.

Ogni file ha un identificativo a basso livello (solitamente un numero), chiamato **inode number.**

Una directory contiene un dizionario che per ogni file fa corrispondere nome “user-readable” e inode number.

Anche una directory ha un proprio identificatore a basso livello → **inode number.**

Posizionando directory all’interno di altre directory è possible creare degli alberi di directory (**directory tree**), tale albero ha sempre una radice (**root directory**).

C 39.3 Creare file

Per creare un file si fa uso della systemcall open() con parametro O_CREAT.

open() ritorna un file descriptor, che corrisponde ad un intero privato per il processo corrente che viene utilizzato per riferirsi al file (può essere visto come una sorta di puntatore).

Per ogni processo, un array tiene traccia dei file aperti (array di file descriptor).

C 39.4 Scrivere e leggere file

Prendiamo in esame le systemcall effettuate dal comando shell “**cat foo”** usato per stampare a schermo il contenuto del file foo.

La prima systemcall è open() con parametro O_RDONLY per aprire il file in sola lettura.

Si noti che open, nell’esempio, ritorna 3 poiché sono già aperti 3 file STDIN, STDOUT, STDERR.

Di seguito viene chiamata read(), systemcall che ha come parametri il file descriptor, buffer in cui salvare quanto letto e dimensione del buffer; viene ritornato il numero di byte letti.

Viene poi chiamata la syscall write() per scrivere quanto appena letto su STDOUT.

Il programma chiama nuovamente read() per leggere altri caratteri, non essendocene altri da leggere ritorna 0.

Infine viene chiamata la systemcall close() per chiudere il file.

C 39.5 Scritture e letture non sequenziali

In alcuni casi è necessario scrivere o leggere in uno specifico punto del file, per farlo si fa uso della systemcall lseek() sposta il puntatore del file alla posizione specificata dall'utente.

L'argomento principale è il file descriptor che viene restituito dalle system call open() e viene utilizzato come riferimento per tutte le successive operazioni su quel file.

La funzione restituisce la nuova posizione del puntatore del file.

Altri argomenti della system call lseek sono:

1. **l'offset** → che indica di quanti byte spostare il puntatore del file (può essere positivo o negativo)
2. **punto di riferimento** → che indica da dove partire per calcolare l'offset

Quest'ultimo può essere:

- SEEK_SET → inizio del file
- SEEK_CUR → posizione corrente del puntatore
- SEEK_END → fine del file

Ad esempio, se l'offset è 100 e il punto di riferimento è SEEK_CUR, lseek() sposterà il puntatore del file di 100 byte verso il fondo rispetto alla posizione corrente del puntatore.

Si noti che nel caso in cui si apra due volte lo stesso file, si otterranno due file descriptor differenti, per tanto gli offset verranno incrementati in maniera indipendente.

File table è una struttura dati utilizzata dal sistema operativo per tenere traccia dei file aperti dai processi.

Ogni volta che un processo apre un file, il sistema operativo crea una voce nella file table per quel file.

Questa voce è chiamata **file table entry** e contiene informazioni sul file, come il file descriptor, l'offset del puntatore del file, il punto di riferimento del file e altri metadati.

In sostanza, la file table è una tabella contenente tutte le informazioni riguardanti i file aperti dai processi in esecuzione, mentre la file table entry rappresenta un singolo record all'interno della file table che contiene informazioni riguardanti un singolo file aperto.

C 39.6 File table entries condivise

In alcuni casi un record della **File Table** può essere condiviso tra più processi, ciò accade per esempio quando un processo genera un processo figlio con la systemcall fork()

La systemcall dup() permette di creare un file descriptor che si riferisce allo stesso file di un file descriptor passato come parametro (crea un duplicato).

C 39.7 Scrivere immediatamente con fsync()

Per ragioni di performance, il file system non scrive immediatamente su disco a seguito di una chiamata a write(), bensì mantiene per un certo tempo le modifiche in un buffer.

Se si desidera forzare la scrittura su disco, si utilizza la systemcall fsync() a cui passare un file descriptor, la funzione ritorna quando tutte la scritture sono state completate.

Per evitare spiacevoli bugs quando si forza la scrittura di un certo file è opportuno forzare la scrittura anche della directory in cui il file è contenuto.

C 39.8 Rinominare i file

Per rinominare un file si utilizza il comando mv, il quale effettua una systemcall a rename() passando due parametri: attuale nome e nuovo nome.

rename() è una funzione atomica, garantisce che in caso di crash del sistema durante la sua esecuzione, al riavvio il file sarà presente con il nome precedente oppure con il nuovo nome (non con nomi “a metà”).

Gli editor di testo, quando si modifica un file (ad esempio foo.txt), solitamente salvano il file con un nome del tipo “foo.txt.tmp”, forzano la scrittura con fsync() ed infine rinominano il file con il nome originale (foo.txt) al fine di evitare perdita di informazioni.

C 39.9 Ricavare informazioni sui file

Il file system mantiene una serie di informazioni per ogni file, tali informazioni sono dette **metadati,** per ricavare tali informazioni si usano le systemcall stat() e fstat().

Tali informazioni comprendono:

- Dimensione (in bytes)
- Identificatore a basso livello (**inode number**)
- informazioni di proprietà
- timestamp di accesso, modifica ecc.

La struttura in cui queste informazioni sono salvate è detta **inode**.

Gli **inode** di ogni file sono salvati su disco, per i file aperti una copia è presente in memoria RAM per ragioni di performance.

C 39.10 Eliminare file

Per eliminare un file si utilizza il comando rm, il quale chiama la systemcall unlink() passando come parametro il nome del file da rimuovere, ritorna 0 ad operazione completata.

Maggiori informazioni riguardo il funzionamento di rm nei paragrafi successivi.

C 39.11 Creare directory

Per creare una directory si utilizza il comando mkdir che chiama l'omonima systemcall mkdir().

Alla creazione una directory è considerata vuota, tuttavia contiene "." un riferimento a se stessa e ".." un riferimento alla directory padre.

C 39.12 Leggere directories

Per leggere il contenuto di una directory si utilizzano le systemcall opendir(), readdir() e closedir().

Per ricavare l’intero contenuto occorre chiamare readdir() iterativamente all’interno di un ciclo.

Ogni oggetto estratto corrisponde ad una struttura contenente: nomefile, inode number, offset per l’oggetto successivo, dimensione, tipo di file.

C 39.13 Eliminare directories

Per eliminare una directory si utilizza il comando rmdir, che chiama l’omonima systemcall rmdir().

Si noti che con tale comando è possibile eliminare solamente directory vuote.

C 39.14 Hard links

La systemcall link() (utilizzata dal comando ln) è utilizzata per creare un collegamento tra un nuovo nome ed un file esistente.

Nella directory viene inserito un nuovo record, con nome differente ma stesso inode number del file originale; si noti che non si tratta in alcun modo di una copia!

Per meglio comprendere il funzionamento di link(), viene illustrato il processo di creazione di un file.

La creazione di un file si compone di due principali fasi:

- Creazione dell’inode.
- Inserimento del record nella directory → si crea quindi un link tra inode e nome del file.

Ogni inode ha un campo **reference count** che indica il numero di collegamenti tra inode e nomi file.

Quando il **reference count** raggiunge 0 (condizione verificata da unlink()) l’inode ed i relativi dati vengono effettivamente eliminati.

Dopo quest’ultima precisazione si dovrebbe aver meglio compreso il funzionamento del comando rm e di conseguenza il processo di eliminazione di un file.

C 39.15 Symbolic Links

Un symbolic link è di fatto un file, si tratta di un terzo tipo di dato conosciuto dal file system (dopo file e directories).

Il contenuto del symbolic link è di fatto il percorso al file collegato.

Se si elimina il file a cui un link è collegato, aprendo il link si genererà un errore.

C 39.16 Bit di permessi e ACL

Il comando ls -l permette di ricavare una serie di informazioni aggiuntive di un elemento tra cui i relativi permessi.

Prendendo ad esempio il seguente output del comando ls -l.

```
-rwxr--r--  1 remzi wheel  0 Aug 24 16:29 foo.txt
```

Il primo carattere “-” indica che si tratta di un regolare file (“d” in caso di directory e “l” per symbolic link).

Le seguenti tre terzine indicano i permessi (Read, Write, execute) rispettivamente per proprietario, gruppo, altri.

Ovviamente il proprietario può in qualsiasi momento modificare i permessi, ad esempio con il comando chmod.

Il bit di execute di una directory permette di cambiare directory (comando cd) e creare file all’interno (solo se anche il bit write è abilitato).

Un metodo più sofisticato per gestire i permessi è l’utilizzo di **Access Control List** adottato da alcuni file system (ad esempio AFS).

C 39.17 Creare e montare un file system

Per creare un file system, gli stessi file system mettono a disposizione un tool “**mkfs**” , che prende in input un device (ad esempio una partizione di un disco) e un tipo di file system.

Una volta creato un file system è necessario **montarlo** per renderlo accessibile, per farlo si usa il comando mount che prende come parametro il **mount point** ovvero una directory a cui collegare la root directory del file system appena creato.

Il montaggio permette di unificare tutti i file system in un singolo albero.

Capitolo 40 File system implementation - Very Simple File System

In questo capitolo introdurremo una semplice implementazione di un file system, il **Very Simple File System (vsfs)**, una versione semplificata di un tipico UNIX file system.

C 40.1 Il modo di pensare

Nello studio dei file system è necessario tenere in considerazione due aspetti:

- Strutture dati utilizzate per organizzare dati e metadati, FS semplici utilizzano array di blocchi o oggetti mentre FS più sofisticati utilizzano strutture basate su alberi.
- Metodi di accesso

C 40.2 Organizzazione generale

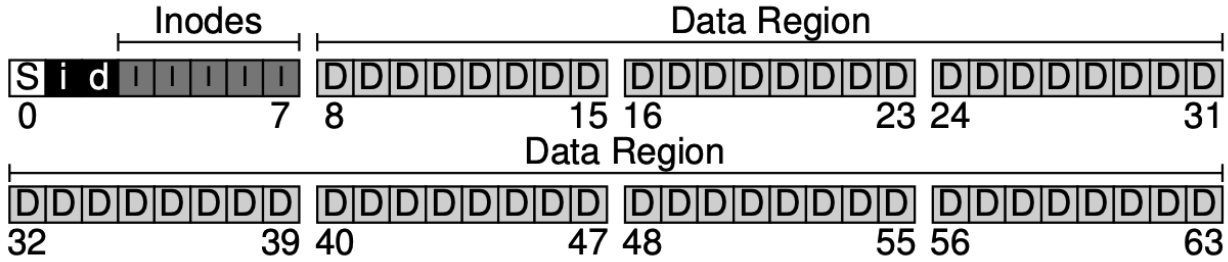
La prima cosa da fare è dividere il disco in blocchi, assumiamo di 4KB ciascuno. Si consideri un disco con 64 blocchi.

Riserviamo una parte (56 blocchi) ai dati utente (**data region**) ed una parte per gli inode, la **inode table** (5 blocchi).

Infine occorre tenere traccia dei blocchi (inode e data) disponibili e già allocati, per farlo si utilizza una struttura semplice detta **bitmap,** in cui ogni bit indica se il corrispondete blocco è libero(0) o occupato (1).

Perciò si utilizza un blocco per **inode bitmap** ed un altro blocco per **data bitmap.**

Il rimanente blocco è detto **superblock** e viene utilizzato per salvare informazioni proprie del file system (ad esempio il numero di blocchi di inode e data ecc.).



C 40.3 Inode (index node)

Una delle più importanti strutture di un file system sono gli inode, ci si riferisce ad un inode con il relativo **i-number.**
In VSFS dato un i-number è possibile calcolare direttamente la sua posizione sul disco.

Di seguito i calcoli per ricavare il numero di blocco e relativo settore

$$blk = (inumber * sizeof(inode_t))/blockSize;$$

$$sector = ((blk * blockSize) + inodeStartAddr)/sectorSize;$$

All'interno dell' inode sono contenuti tutti i metadati del file → tipo, informazioni di accesso/permessi, timestamp creazione, modifica, ultimo accesso e informazioni relative alla posizione del data block.

Un approccio semplice consiste nel salvare nell' inode puntatori diretti agli indirizzi di disco dei relativi data block, ma se si immagina un file di enormi dimensioni, diviso su numerosi blocchi, i cui puntatori non possono essere tutti contenuti nell'inode, allora ci si trova in una condizione spiacevole.

Multi-Level Index

Per ovviare al problema sopracitato è possibile utilizzare un puntatore indiretto, che punti ad un blocco contenente a sua volta una serie di puntatori ai blocchi di dati.

Tale approccio è ovviamente estendibile a vari livelli.

Il multi level index potrebbe sembrare un approccio non ottimale, in effetti lo è; tuttavia ricerche hanno dimostrato che mediamente la dimensione dei file non è eccessiva e

quindi le performance rimangono buone.

C 40.4 Organizzazione delle directory

Una directory può essere vista come una lista di coppie nomefile e inode-number; tuttavia è necessario anche memorizzare la lunghezza della stringa nomefile.

Quando si cancella un file contenuto in una directory, il campo lunghezza viene posto ad uno specifico valore (per esempio 0) per indicare che quell'elemento della lista può essere sovrascritto.

Su disco una directory è del tutto simile ad un file, ha il proprio inode e i propri blocchi di dati.

C 40.5 Gestione dello spazio libero

Come anticipato in precedenza per tenere traccia dei blocchi liberi VSFS utilizza due bitmap.

Alcuni FS linux garantiscono, inoltre, che una porzione del file sia posizionata in blocchi contigui su disco al fine di migliorare le performance.

C 40.6 Metodi di accesso: Lettura e Scrittura

Lettura dal disco

Nel seguente esempio si assume di voler leggere il file "/foo/bar" la cui dimensione corrisponde a 3 blocchi.

La prima systemcall è open("/foo/bar", O_RDONLY) il sistema deve, partendo dall' i-number della **root directory** noto a priori (2), ricavare tutti gli inode da attraversare.

Quindi le operazioni eseguite sono nell'ordine:

1. Lettura inode radice → ricavo data address
2. Lettura root data → ricavo inode number "foo"
3. Lettura inode "foo" → ricavo data address

4. Lettura “foo” data → ricavo inode number “bar”
5. Lettura inode “bar” → ricavo data address

Al termine viene generato e ritornato il file descriptor.

Ora che il file è stato correttamente aperto viene chiamata read(), la quale ad ogni chiamata oltre a leggere i dati, aggiorna il campo dell’ inode relativo all’ultima modifica e il campo offset dell’open file table.

Infine il file viene chiuso senza che si necessiti di ulteriore I/O.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			**read**	**read**	**read**	**read**	**read**			
read()					**read** **write**			**read**		
read()					**read** **write**				**read**	
read()					**read** **write**					**read**

Figure 40.3: **File Read Timeline (Time Increasing Downward)**

Scrittura sul disco

Il processo di scrittura prevede in primis l’apertura del file come descritto precedentemente.

In seguito oltre alla scrittura è necessario allocare lo spazio necessario ad ospitare i nuovi dati, ciò corrisponde a modifiche del bitmap e dell’ inode.

Sono necessarie in tutto 5 chiamate I/O:

- due per leggere e scrivere data bitmap.
- due per leggere e scrivere inode.
- una per la effettiva scrittura.

Quando l’operazione di scrittura comporta la creazione del file il numero di chiamate I/O sale a 10.

Di seguito uno schema riassuntivo riguardo la scrittura con creazione del file.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		**read write**	**read**	**read**	**read write**	**read**	**read**			
				write			**write**			
write()	**read write**				**read**			**write**		
					write					
write()	**read write**				**read**				**write**	
					write					
write()	**read write**				**read**					**write**
					write					

Figure 40.4: **File Creation Timeline (Time Increasing Downward)**

C 40.7 Caching and Buffering

Per ovviare a problemi di performance molti file systems fanno un uso importante della memoria di sistema (RAM) come cache.

I primi file system usavano **fixed-size cache** per mantenere i blocchi più richiesti, solitamente corrispondeva a circa il 10% della RAM.

Moderni file system invece utilizzano un **dynamic partitioning** per risparmiare spazio (FS richiede la quantità di memoria da allocare).

Per migliorare le prestazioni della scrittura i file system fanno largo uso del buffering, infatti può essere utile ritardare l'operazione per :

- Agglomerare scritture sullo stesso blocco (si pensi alla creazione di due file, quindi le scritture sul blocco bitmap possono avvenire contemporaneamente).
- Schedulare le scritture in maniera più efficiente.
- Evitarle completamente (si pensi al caso in cui si crei un file e subito dopo lo si cancelli)

Per questi motivi i moderni file system scrivono effettivamente in memoria ogni 5-30 secondi.

Tuttavia alcune applicazioni (ad esempio i database) non gradiscono tale approccio, perciò fanno spesso uso della systemcall fsync() oppure fanno uso di file system appositamente progettati.

Capitolo 41 Fast File System

Inizialmente il file system di linux era molto semplice, aveva una struttura simile a quella del VSFS.

C 41.1 Il problema di performance

Purtroppo le performance erano terribili, infatti il disco veniva utilizzato al pari di una memoria RAM, i dati erano sparsi su tutto il disco, di conseguenza i tempi di posizionamento erano proibitivi.

Il disco risultava frammentato, in quanto non veniva attuata una gestione dello spazio libero, di conseguenza file logicamente vicini risultavano sparsi nel disco.

Un secondo problema era il **block size** (512 Bytes) troppo piccolo, buono per ridurre la internal fragmentation, ma pessimo per il trasferimento poiché ogni 512B era

necessario un riposizionamento.

C 41.2 FFS

Per ovviare ai problemi del primo file system, viene progettato **Fast File System** al fine di migliorare le performance. Gli sviluppatori hanno voluto mantenere la stessa interfaccia ma modificare l'implementazione interna rendendola "disk aware".

C 41.3 Organizzazione: il gruppo di cilindri

FFS divide il disco in gruppi di cilindri :

- un cilindro è una serie di tracce su differenti superfici del hard drive, tutte con la stessa distanza dal centro.
- Un gruppo è un insieme di N cilindri consecutivi.

I moderni hard disk, in realtà, nascondono la propria "geometria" e rendono disponibile al file system solamente un address space logico.

FFS garantisce che l'accesso a due file nello stesso gruppo non comporti lunghi tempi di attesa.

La struttura di FFS è del tutto simile a quella di VSFS, è presente un superblock, bitmap per inode e dati, segmento inode e segmento dati.

C 41.4 Come allocare file e directory

FFS per allocare una directory sceglie un gruppo con poche directory già allocate ed un elevato numero di inode liberi per poter allocare i file.

Per allocare i file invece, si accerta di allocare i blocchi di dati nello stesso gruppo dell'inode e posiziona i file all'interno dello stesso gruppo della directory in cui sono.

C 41.5 Misure

La metrica utilizzata per valutare le performance consiste nel valutare quante directory si debba percorrere prima di raggiungere il comune antenato di due file.

Studi dimostrano che FFS abbia buone performance.

Paragrafo poco chiaro quindi in parte omesso.

C 41.6 Eccezione dei grandi file

La politica generale di FFS non funziona bene su files di grandi dimensioni.

I file di grandi dimensioni vanno ad occupare quasi tutti i blocchi di un certo gruppo, perciò quando si desidera inserire un nuovo file nella stessa directory lo si dovrà posizionare in un altro gruppo, ciò comporta un calo di performance.

FFS prevede quindi una **large file exception** per cui un file di grandi dimensioni viene suddiviso su più gruppi.

Si può intuire che anche quest'ultimo approccio determina un calo di performance poiché nella lettura del file di grandi dimensioni si verificano numerosi riposizionamenti della testina.
Per ovviare a questo problema occorre **ammortizzare,** quindi se si aumenta la dimensione dei chunk, si può trasferire più dati in un'unica operazione di lettura/scrittura, il che può aiutare a ridurre il numero di operazioni di seek (posizionamento testina) necessarie per accedere a tutti i dati di un file di grandi dimensioni.

Un esempio

Si immagini un seek time = 10 ms, un disco che trasferisce dati a 40MB/s. Se si desidera usare 50% del tempo per seek e il restante 50% per il trasferimento, allora il trasferimento deve durare 10ms per ogni seek.

Si procede quindi con il seguente calcolo:

$$\frac{40MB}{s} * \frac{1024KB}{1MB} * \frac{1s}{1000ms} * 10ms = 409.6KB$$

Aumentando la percentuale del tempo di trasferimento si ottengono chunk size via via maggiori.

FFS non svolge tutti questi calcoli, bensì posiziona i primi 12 blocchi nello stesso gruppo dell' inode, i restanti in gruppi differenti.

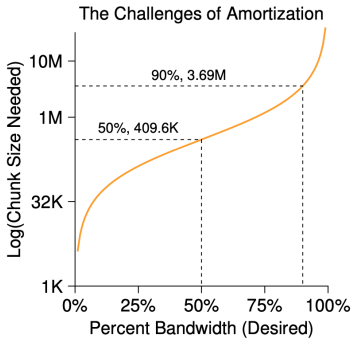


Figure 41.2: **Amortization: How Big Do Chunks Have To Be?**

C 41.7 Altri dettagli su FFS

FFS usa un particolare layout dei settori per ottimizzare le performance, i settori sequenziali sono alternati tra di loro, così che in caso di letture sequenziali non si dovrà attendere il tempo di un intero giro per posizionare il settore richiesto sotto la testina (ipotizzando che la testina stia leggendo il blocco 1 al termine della lettura si potrebbe trovare a metà del blocco due, a causa della rotazione) ; questa tecnica è detta **parametrization.**

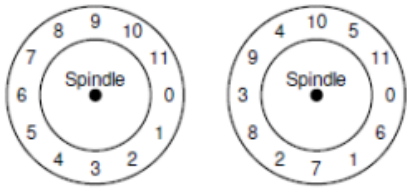


Figure 41.3: **FFS: Standard Versus Parameterized Placement**

Il problema viene oggi risolto anche attraverso l'utilizzo di cache.

FFS è stato il primo file system a supportare filename lunghi (a dimensione variabile) e ad introdurre i symbolic link.

Capitolo 42

Le strutture dati di un file system devono ovviamente essere persistenti, pertanto è necessario aggiornare tali strutture tenendo conto di possibili crash di sistema o perdite di tensione.

C 42.1 Un esempio

Si immagini una struttura come quella dei capitoli precedenti (bitmap, inode, dati), una richiesta di scrittura di dati comporta di fatto al più 3 scritture su disco: una del bitmap, una dell' inode e una dei dati; un crash può avvenire durante ciascuna delle tre fasi.

Scenari possibili

Si immagini che una sola delle tre scritture sia effettivamente stata eseguita:

- Solo il data block è stato scritto: nessun inode punta a quel blocco ed il relativo bitmap non è aggiornato, è come se la scrittura non fosse avvenuta.
- Solo l'inode è stato scritto: l'inode punta ad un data block che non è stato scritto, si rischia di leggere dei dati non pertinenti; inoltre il bitmap indica che il blocco dei dati non è allocato siamo in una condizione di **file system inconsistency.**
- Solo il bitmap è stato scritto: nessun inode punta al blocco occupato nel bitmap, quel blocco seppur di fatto libero non verrà mai usato.

Si immagini ora che due delle tre scritture siano state effettivamente eseguite:

- Inode e bitmap sono stati scritti: Si rischia di leggere dati non pertinenti perché non sono mai stati scritti.
- Inode e dati sono stati scritti: c'è **inconsistency** tra inode e bitmap, tale condizione deve essere risolta prima di usare il file system.
- Bitmap e dati sono stati scritti: non si può sapere a quale file appartengano i dati.

Una soluzione potrebbe essere di rendere le operazioni di scrittura atomiche, ma purtroppo ciò non è possibile farlo con un disco a causa della sua struttura.

C 42.2 Soluzione 1: Il File System Checker

Tale approccio si pone come obiettivo quello di risolvere eventuali **crash-inconsistency** al reboot; un esempio è rappresentato dal tool linux **fsck**.

FSCK tuttavia non può risolvere tutti i casi possibili, il caso in cui l'inode è presente ma punta a dati errati non è riconosciuto.

Il tool viene eseguito prima di montare il file system, di seguito le principali fasi di funzionamento.

- **Superblock:** in prima istanza fsck controlla il superblock (ad esempio verifica che la dimensione sia maggiore del numero di blocchi ecc.)
- **Free blocks:** Itera su tutti i blocchi, al fine di costruite una bitmap corretta, la confronta con quella presente ed eventualmente la corregge.
- **Inode state:** verifica l'integrità di ogni inode, se ha "sospetti" elimina l'inode e aggiorna bitmap.
- **Inode links**: calcola il link counter di ogni inode (partendo dalla radice verifica ogni directory alla ricerca di riferimenti ad un inode). Se il numero è diverso lo aggiorna, se viene trovato un inode senza riferimenti ad esso, viene messo nella directory "lost+found".
- **Duplicati:** se due inode puntano allo stesso blocco, uno viene eliminato oppure si crea una copia dei dati così i due inode punteranno a blocchi differenti.
- **Bad blocks:** Un puntatore è considerato "bad" se punta ad un blocco che non esiste o maggiore della dimensione della partizione.
  Il puntatore sbagliato verrà rimosso dall'inode.
- **Directory checks:** controlla che le directory siano ben formattate (ogni file contenuto sia effettivamente allocato, esistenza di "." e "..").

Ovviamente tale approccio ha un problema, è TROPPO LENTO!

C 42.3 Soluzione 2: Journaling

L'idea del **journaling** (anche noto come **write-ahead logging**) è quella di tenere nota delle scritture che stanno per essere eseguite prima di eseguirle effettivamente.

Di seguito si illustra **Linux ext3** un file system che fa uso di journaling, tale implementazione prevede che una parte del disco venga riservata per il **journal.**

Data journaling

Ipotizzando di voler modificare bitmap, inode e dati; tali modifiche vengono dapprima scritte nel journal, precedute da una intestazione e un terminatore che contiene il **transaction identifier** (TID).

Una volta salvati i tre blocchi nel journal è possibile aggiornare i blocchi veri e propri, tale fase è detta **checkpointing.**

Questa prima implementazione può quindi essere riassunta in 2 fasi:

- Journal write
- Checkpoint

Nel caso in cui un crash del sistema avvenga durante una delle operazioni di scrittura del journal, si va incontro a dei problemi, un soluzione potrebbe essere quella di scrivere le parti del journal una alla volta ma ciò richiederebbe troppo tempo; perciò si preferisce una singola scrittura sequenziale.

Tuttavia il disco non garantisce che le scritture vengano effettuate nell'ordine dato, ci si potrebbe quindi trovare, per esempio, ad avere scritto correttamente intestazione, terminatore, bitmap e inode ma non i dati. In questo caso, al reboot, verrebbero ripristinati dati inconsistenti.

Per evitare quanto appena descritto, la scrittura del terminatore viene effettuata solo al termine della scrittura dei blocchi e dell'intestazione.

Le fasi possono essere ora riassunte in:

- Journal write
- Journal commit (scrittura del terminatore)
- Checkpoint

Recovery

Per ripristinare le eventuali modifiche perse a causa di un crash, è necessario semplicemente eseguire le scritture presenti nel journal (in ordine).

Rendere il Log finito

Lo spazio riservato al journal è ovviamente finito, un metodo per gestirne la dimensione è quello di riutilizzare lo spazio, rendendolo quindi una sorta di coda (**circular log**).

Una parte del journal detta **Journal superblock** viene riservata per indicare le posizioni libere del Journal.

Le fasi possono ora essere riassunte in:

- Journal write
- Journal commit
- Checkpoint
- Free: si modifica il journal superblock per segnalare lo spazio libero.

Metadata journaling

Il data journaling ha un grave difetto, le operazioni di scrittura vengono di fatto raddoppiate e di conseguenza anche i tempi.

Un approccio migliore è il **metadata journaling,** il quale prevede che vengano scritti nel journaling solo i blocchi relativi a bitmap e inode (oltre a intestazione e terminatore), mentre il blocco di dati viene direttamente scritto sul file system vero e proprio.

In questo modo si riducono notevolmente i tempi, proprio perché l’elemento più pesante da scrivere sono ovviamente i dati.

Per il corretto funzionamento di tale tecnica è fondamentale che l’aggiornamento dei dati avvenga prima della scrittura sul journal.

L’ordine delle operazioni è il seguente:

- Data write
- Journal metadata write
- Journal commit
- Checkpoint metadata
- Free

Il metadata journaling è di fatto il journaling più utilizzato (ad esempio WIN NTFS o linux EXT3 il quale permette di scegliere entrambe le modalità).

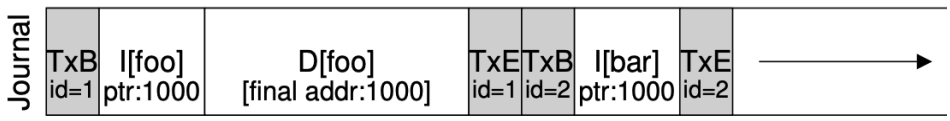
In realtà non è indispensabile che la prima operazione sia la scrittura dei dati, l'importante è che avvenga prima del journal commit.

Caso particolare

Si consideri il caso in cui si modifichi una directory "foo" (per esempio creando un file all'interno), nel journal verranno inseriti i blocchi dati e inode (si inseriscono anche i dati poiché una directory è considerata come un metadato).

Successivamente si elimina tale directory e si crea un nuovo file "bar" (in una directory qualsiasi), il file system decide di assegnare al nuovo file lo stesso blocco dati precedentemente assegnato alla directory. I dati vengono quindi scritti direttamente e successivamente si aggiorna il journal e a seguire i passaggi successivi.

Si immagini che ci sia un crash del sistema, al reboot si provvederà nell'ordine a riscrivere inode della directory, dati della directory e a seguire aggiornare lo stesso inode con le informazioni relative al file "bar", ciò comporta quindi una completa perdita dei dati del file "bar".



Per ovviare al problema appena citato si aggiunge un flag "**revoke**" ad ogni record del journal da abilitare in caso di eliminazione della directory.

C 42.4 Soluzione 3: Altri approcci

Esistono altri approcci per assicurare persistenza, di seguito una breve descrizione dei principali.

- **Soft Updates**: ordina le scritture alle strutture in modo da non avere mai inconsistenza. Necessita di piena conoscenza del file system e aumenta di molto la complessità del sistema
- **Copy on write:** non aggiorna mai direttamente i file o directory, ma aggiorna delle copie. Dopo un certo numero di aggiornamenti sposta la radice del file system alla sezione con le copie aggiornate.
- **Backpointer-based consistency:** Si aggiunge un puntatore “all’ indietro” in ogni blocco (ad esempio un blocco dati ha un puntatore al relativo inode).
- **Optimistic crash consistency:** si eseguono quante più scritture possibili, poi si esegue una sorta di checksum. Un metodo molto performante ma che necessita di una interfaccia del disco differente.