

Sistemi Operativi

Ionut Zbirciog

5 October 2023

1 Cos'è un Sistema Operativo?

Un software che si interfaccia con i componenti hardware della macchina (CPU, MEMORIA, MEMORIE NON VOLATILI, DISPOSITIVI I/O).

2 Componenti di un calcolatore moderno

- Uno o più processori
- Memoria principale
- Dischi e unità flash
- Periferiche I/O

2.1 Doppia modalità supportata dall'hardware

Un'applicazione è un processo che esegue delle istruzioni. Il sistema mette a disposizione delle funzioni (chiamate di sistema). Il sistema operativo si prende carico di gestire le risorse ed eseguire le istruzioni.

- Modalità kernel (supervisor)
- Modalità utente

3 Il SO come una macchina

- Idea di Astrazione: SO si pone tra l'hardware e le applicazioni (l'obiettivo del SO è di astrarre l'hardware).
- Visione top-down: SO fornisce astrazioni ai programmi applicativi.
- Vista Bottom-up: SO gestisce parti di un sistema complesso, fornisce un'allocazione ordinata e controllata delle risorse.

4 Il SO come gestore di risorse

Il SO permette di gestire le risorse per:

- Eseguire più programmi in esecuzione
- Supportare più utenti

Il multiplexing permette di mettere a disposizione delle risorse in modo condiviso, sia nel tempo (CPU, Stampante) che nello spazio (Memoria centrale, Disco).

5 Breve Storia dei Sistemi Operativi

1. First Generation: Vacuum Tubes

- Tubi vuoti che emettevano luce per simulare gli 1 e gli 0.
- Nessun supporto per la programmazione; l'unico modo era spostare i connettori.
- I calcoli potevano essere eseguiti uno alla volta (limitata ottimizzazione della macchina).

2. Second Generation: Transistors and batch systems

- Scomposizione delle operazioni della macchina e del sistema operativo in blocchi operativi.
- Spiegazione delle operazioni batch.
- Componenti come il lettore di schede (per programmi, compilatori e il sistema operativo) e il lettore di nastri (da schede a nastri).
- I programmi e i dati da elaborare venivano trascritti e inviati all'esecuzione.

3. Third Generation: Integrated Circuits and Multiprogramming

- Caricamento di applicazioni contemporaneamente grazie a memorie più grandi.
- Partizionamento della memoria.
- Spooling: caricamento dei lavori senza interruzioni.
- Time Sharing: la CPU viene assegnata a lavori o utenti diversi mentre è in attesa.
- Non è stata implementata fino agli anni '70 a causa della mancanza di protezione hardware per garantire che, in caso di errore, un processo non scrivesse in un'area riservata ad un altro.

4. Fourth Generation: Personal Computer

5. Mobile Computers: Smartphone

6 UNIX

SO multiutente e multiprogrammazione, open-source. Il linguaggio C fu sviluppato per scrivere UNIX. Standard POSIX sviluppato dall'IEEE per garantire compatibilità inter-sistemi. Da UNIX derivano molti altri sistemi operativi.

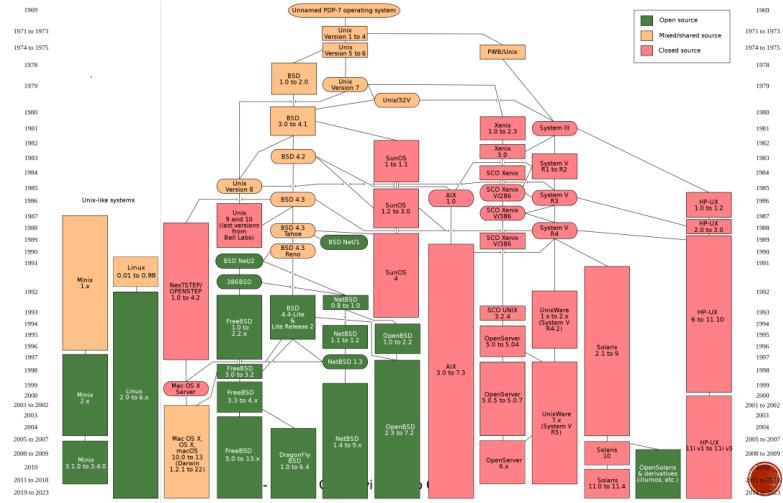


Figure 1: Sistemi operativi derivati di UNIX

6.1 MINIX - una variante di UNIX

Un piccolo sistema scritto da Tanenbaum, compatibile con gli standard UNIX.

6.2 Da MINIX a LINUX

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) A T clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and g c c(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)

P.S. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than A T-harddisks, as that's all I have :-)

Figure 2: Il messaggio di Linus Torvalds

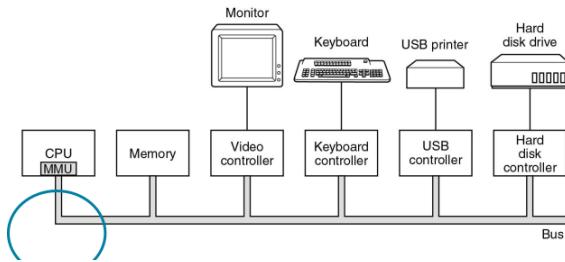


Figure 3: Architettura di un calcolatore

7 Uno sguardo all'hardware

7.1 Il processore

Il ciclo della CPU: preleva (fetch), decodifica (decode), esegue (execute).

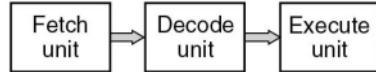


Figure 4: fetch - decode - execute

- Registri
 - Program Counter: indica l'istruzione successiva
 - IR: indica l'istruzione che viene eseguita
 - Stack Pointer: punta alla cima dello stack della memoria
 - Program Status Word (PSW): contiene informazioni sullo stato del programma, fondamentale per chiamate di sistema e I/O.
- Multiplexing: il sistema operativo esegue programmi in modo efficiente
- Pipeline: Esegue in parallelo istruzioni che possono essere eseguite a livello circuitale. Anche in caso di un'istruzione condizionale, la pipeline esegue anche l'operazione successiva pur di non fermarsi dall'eseguire istruzioni.
- Più di un processore: Più processori fisici o logici, multithreading.

7.2 La memoria di un calcolatore

Problemi del sistema cache:

- Quando inserire un nuovo elemento nella cache?
- In quale riga della cache inserire il nuovo elemento?

- Quale elemento rimuovere dalla cache quando è necessario uno slot.
- Deve mettere un elemento appena eliminato nella memoria più grande.

Nota: Più la memoria è veloce, più è piccola

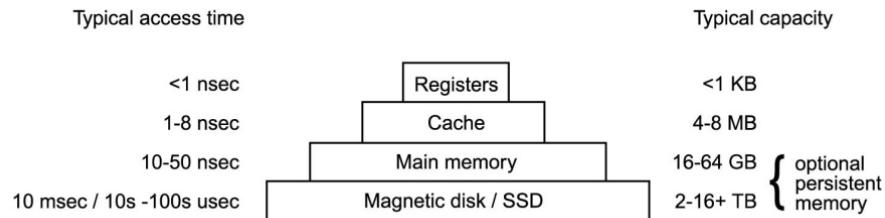


Figure 5: Gerarchia della memoria

7.3 Dispositivi di I/O

- Il controller: più semplice da usare per il SO, ogni controller ha bisogno di un driver.
- Il dispositivo: interfaccia elementare ma complicata da pilotare

Driver: pezzetto di sistema operativo che non viene fornito con il sistema operativo, deve essere inserito all'interno del sistema operativo e avere tutti i diritti del sistema operativo. L'unico momento in cui il sistema operativo concede al driver tutti i diritti è al momento dell'avvio, infatti dopo aver installato un driver, il sistema operativo chiede all'utente di riavviare il sistema. In Linux hanno dovuto forzare questa cosa a causa delle porte USB.

Il driver interagisce con il controller per:

- Eseguire l'I/O
 - il processo esegue la chiamata di sistema
 - il kernel effettua una chiamata al driver
 - il driver avvia l'I/O
- Interrogare il dispositivo per vedere se ha finito oppure chiede al dispositivo di generare un interrupt quando ha finito

7.4 Il DMA

DMA (Direct Memory Access) consente ai componenti di accedere direttamente alla memoria del computer senza coinvolgere la CPU, migliora l'efficienza ed aumenta le prestazioni nelle operazioni di I/O.

7.5 Buses

- Dispositivi legacy collegati a un processore hub separato
- USB è stato sviluppato per connettere dispositivi lenti al computer
- La USB deve sia alimentare che comunicare

7.6 Avvio del sistema

Quando si accende la macchina, si legge la ROM con le istruzioni per tutte le periferiche. Il BIOS esegue i comandi e da retta al BOOTLOADER che legge il sistema operativo in base alla periferica.

Sistemi Operativi

Ionut Zbirciog

10 October 2023

1 Sistema Operativo

Un sistema operativo è un software essenziale che gestisce l'hardware di un computer e fornisce servizi fondamentali per l'esecuzione di programmi applicativi. Uno dei ruoli principali di un sistema operativo è mettere a disposizione l'hardware attraverso le chiamate di sistema. Esistono vari tipi di sistemi operativi, ognuno progettato per scopi specifici, tra cui:

- **Sistemi Operativi per Mainframe:** Progettati per gestire sistemi informatici di grande scala, come mainframe aziendali.
- **Sistemi Operativi per Server:** Ottimizzati per fornire servizi e risorse su reti e su Internet.
- **Sistemi Operativi per Personal Computer:** Utilizzati su computer desktop e laptop per l'uso quotidiano.
- **Sistemi Operativi per Smartphone e Computer Palmari:** Progettati per dispositivi mobili.
- **Sistemi Operativi per IoT e Sistemi Operativi Embedded:** Utilizzati in dispositivi embedded e nell'Internet delle Cose.

Ciò che tutti i sistemi operativi hanno in comune sono due aspetti chiave:

- **Extended Machine:** Forniscono un'estensione delle funzionalità e un'astrazione dell'hardware, consentendo ai programmi applicativi di interagire con l'hardware in modo uniforme.
- **Resource Machine:** Gestiscono la protezione e la condivisione equa delle risorse hardware tra i processi in esecuzione.

2 Processo

Un processo è un'entità fondamentale in un sistema operativo. Può essere definito come un programma in esecuzione. I processi rappresentano un'astrazione a livello utente che permette l'esecuzione di programmi. Ad ogni processo è associato uno spazio di indirizzamento, che contiene il codice del programma, i dati e altre informazioni necessarie per l'esecuzione.

Ogni processo ha un identificatore univoco chiamato **PID (Process ID)**, che viene utilizzato per identificarlo in modo univoco nel sistema.

Il layout di base di un processo include le seguenti sezioni di memoria:

- **Stack:** Contiene i puntatori ai dati e viene utilizzato per la gestione delle chiamate di funzioni e delle variabili locali.
- **Data:** Contiene le variabili del programma, incluse le variabili globali e i dati inizializzati.
- **Text:** Contiene il codice eseguibile del programma.
- **Gap:** Fornisce spazio per l'allocazione e la deallocazione dinamica della memoria durante l'esecuzione del processo.

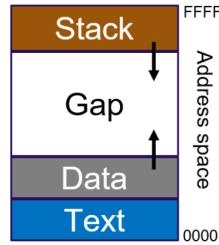


Figure 1: Layout di base di un processo

Il ciclo di vita di un processo comprende diverse fasi, e le informazioni sui processi attivi vengono mantenute in una tabella dei processi attivi. Un processo può essere creato (allocato nella lista dei processi), terminato (deallocato dalla lista dei processi), messo in pausa (spesso i processi sono in uno stato di sospensione) e ripreso. Inoltre, un processo può creare un altro processo, stabilendo una relazione padre-figlio, che definisce come i processi vengono generati.

Il possesso di un processo è assegnato a un utente identificato da un UID (User ID). Un processo figlio ha lo stesso UID del processo padre. Gli utenti possono appartenere a gruppi identificati da un GUID (Group ID). Inoltre, esiste un processo speciale noto come superuser o root, che ha privilegi elevati nel sistema.

3 File

Un file è un'astrazione di un dispositivo di memorizzazione, come un disco. I file possono essere letti e scritti specificando una posizione e una quantità di dati da trasferire. I file sono organizzati all'interno di directory, e le directory stesse possono contenere un elenco di identificatori per i file che contengono. Le directory e i file sono organizzati in una gerarchia, con la radice ("/") come directory principale. In un sistema UNIX, esistono due tipi di percorsi per identificare un file:

- **Percorso Assoluto:** Inizia dalla radice ("/") e specifica la posizione completa del file, ad esempio, "/home/username/file.txt".
- **Percorso Relativo:** È specifico alla directory corrente e si riferisce ai file relativamente alla directory in cui ci si trova, ad esempio, "../documents/report.pdf".

I dischi nei sistemi UNIX si trovano spesso in /mnt/disc o altre posizioni specifiche. In sistemi Windows, i dischi sono associati a lettere dell'alfabeto, come "C:" per il disco principale.

I diritti di accesso a un file sono controllati tramite tuple di tre bit per proprietario, gruppo e altri utenti. I bit di diritti di accesso includono "r" per la lettura, "w" per la scrittura e "x" per l'esecuzione. Ad esempio, "-rwxr-x-x" indica che il proprietario ha il permesso di eseguire, modificare e leggere il file, il gruppo può leggerlo ed eseguirlo, mentre altri utenti possono solo eseguirlo.

La prima lettera di una riga di diritti di accesso può essere "b" o "c" in caso di file speciali di blocco o carattere, ad esempio, per i dischi o le porte seriali.

I file e le pipe sono due tipi di astrazioni di file. Le pipe sono strutture dati che consentono ai processi di comunicare attraverso un canale FIFO (First In First Out).

Terminologia importante legata ai file include il percorso (path), la directory (folder), la directory di lavoro (working directory), il descrittore di file (file descriptor), i file speciali di blocco/carattere e le chiamate di sistema per la gestione dei file.

Termini importanti

- Path
- Folder/Directory
- Working Directory
- File Descriptor
- Block/Character special files
- Pipe

4 Chiamate di Sistema

Le chiamate di sistema rappresentano l'interfaccia che il sistema operativo offre alle applicazioni per richiedere servizi.

Poiché le chiamate di sistema sono specifiche del sistema operativo e dell'hardware, è essenziale che siano efficienti. Per rendere più agevole l'interazione con il sistema operativo, le chiamate di sistema sono spesso incapsulate in librerie C, come libc. Ogni chiamata di libreria corrisponde a una chiamata di sistema specifica. Ad esempio, la chiamata alla funzione `read(file, &buffer, nbyte)` consente di leggere dati da un file, specificando il puntatore al file, il buffer in cui scrivere i dati letti e la quantità di byte da leggere.

Quando un processo in modalità utente ha bisogno di un servizio di sistema, deve eseguire una **trap** o un'istruzione che effettua una chiamata di sistema. Questo passaggio consente al controllo di passare dalla modalità utente alla modalità kernel, dove il sistema operativo può gestire la richiesta. Dopo l'esecuzione di una chiamata di sistema, il controllo ritorna all'istruzione successiva a quella chiamata di sistema.

Le chiamate di sistema seguono una sequenza di passi:

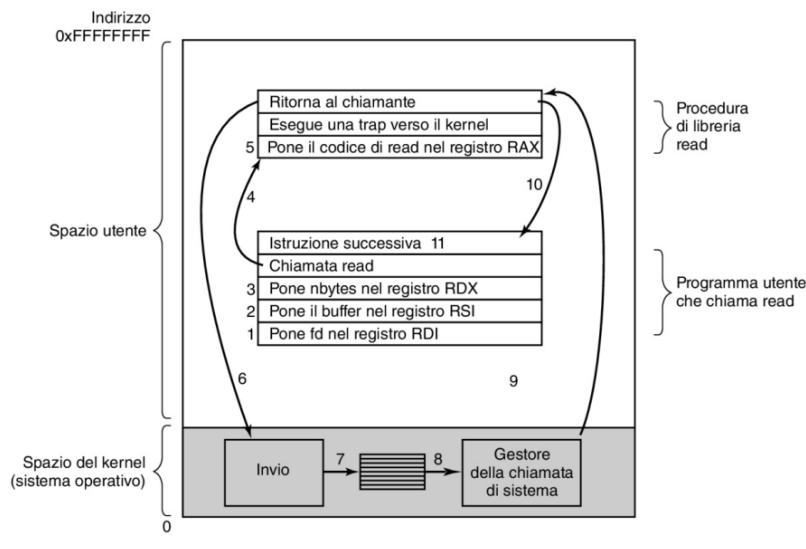


Figure 2: Chiamata di sistema

Esecuzione della chiamata di sistema read(fd, &buffer, nbytes).

1. Per prima cosa, il sistema operativo prepara i parametri mettendoli in registri appropriati, secondo la convenzione di chiamata System V. Da 1 a 3 il sistema mette i parametri in 3 registri.
2. Poi viene l'effettiva chiamata alla procedura di libreria (passaggio 4). Questa istruzione è la normale istruzione di chiamata di procedura usata per chiamare tutte le procedure.
3. La procedura di libreria, che può essere scritta in linguaggio assembly, colloca in genere il numero della chiamata di sistema in un registro noto al sistema operativo, come il registro RAX (passaggio 5). Poi esegue un'istruzione trap (come l'istruzione x86-64 SYSCALL) per passare da modalità utente a modalità kernel.
4. Al passaggio 6, avviene l'esecuzione all'interno del kernel.
5. Il kernel esamina il numero di chiamata di sistema e la smista al gestore corretto.
6. Il gestore della chiamata di sistema esegue il lavoro richiesto.
7. Dopo il completamento del gestore, il controllo può ritornare alla procedura di libreria nello spazio utente. La chiamata di sistema può bloccare il programma chiamante se necessario, ad esempio, se il programma cerca di leggere da una tastiera senza input disponibile.
8. Questa procedura ritorna poi al programma utente nel modo consueto (passaggio 10)
9. Il programma passa alla successiva istruzione (passo 11).

Alcune chiamate di sistema comuni includono quelle per la gestione dei processi, dei file e del file system, nonché il controllo del tempo.

4.1 Chiamate di Sistema per la Gestione dei Processi

- `pid fork()`: Crea un processo figlio identico al processo genitore e restituisce il PID del processo figlio.
- `pid waitpid(pid, &statloc, options)`: Attende che un processo figlio specificato con il PID termini e restituisce lo stato di uscita del processo figlio.
- `s = execve(name, argv, environp)`: Sostituisce l'immagine centrale del processo con un nuovo programma specificato da `name`, passando gli argomenti in `argv` e l'ambiente in `environp`.
- `exit(status)`: Termina l'esecuzione del processo corrente e restituisce uno stato specificato da `status`.

4.2 Chiamate di Sistema per la Gestione dei File

- `fd = open(file, how, ...)`: Apre un file specificato da `file` in modalità lettura, scrittura o entrambe, restituendo un descrittore di file `fd`.
- `s = close(fd)`: Chiude un file aperto identificato dal descrittore di file `fd`.
- `n = write(fd, &buffer, nbytes)`: Scrive dati dal buffer `buffer` in un file identificato dal descrittore di file `fd`, scrivendo `nbytes` byte.
- `n = read(fd, &buffer, nbytes)`: Legge dati da un file identificato dal descrittore di file `fd` nel buffer `buffer`, leggendo un massimo di `nbytes` byte.
- `p = lseek(fd, offset, whence)`: Sposta il puntatore di lettura/scrittura in un file identificato dal descrittore di file `fd` in base all'offset specificato e alla modalità di spostamento `whence`.
- `s = stat(name, &buf)`: Ottiene informazioni sullo stato di un file specificato da `name` e le memorizza nella struttura `buf`.

4.3 Chiamate di Sistema per la Gestione del File System

- `s = mkdir(name, mode)`: Crea una nuova directory con il nome specificato da `name` e con i diritti di accesso specificati da `mode`.
- `s = rmdir(name)`: Rimuove una directory vuota con il nome specificato da `name`.
- `s = link(name1, name2)`: Crea un riferimento a un file specificato da `name1` con un nome alternativo specificato da `name2`.
- `s = unlink(name)`: Rimuove una voce dalla directory specificata da `name`, eliminando il file associato.

- `s = mount(special, name, flag)`: Monta un file system con una specifica opzione di montaggio identificata da `flag` sul punto di montaggio specificato da `name`.
- `s = umount(special)`: Smonta un file system identificato da `special` dal punto di montaggio.
- `s = chdir(dirname)`: Cambia la directory di lavoro corrente del processo in quella specificata da `dirname`.
- `s = chmod(name, mode)`: Modifica i bit di protezione di un file specificato da `name` in base ai diritti di accesso specificati da `mode`.
- `s = kill(pid, signal)`: Invia un segnale specificato da `signal` a un processo identificato dal PID specificato da `pid` (nota: questa chiamata di sistema non termina il processo, ma invia un segnale ad esso).
- `s = time(&seconds)`: Restituisce il tempo trascorso in secondi dal 1 gennaio 1970 e lo memorizza nella variabile `seconds`.

5 Struttura di un Sistema Operativo

La struttura di un sistema operativo può variare a seconda dell'approccio adottato. Un'approccio comune è il modello monolitico, in cui l'intero sistema operativo è eseguito come un unico programma in modalità kernel. In questo modello, il sistema operativo è scritto come una raccolta di procedure collegate all'interno di un unico grande programma binario eseguibile. Questo approccio fornisce elaborazioni efficienti, ma può rendere il sistema operativo pesante e complesso. Inoltre, un errore in una parte del sistema può influenzare altre parti, causando crash sistematici.

Una struttura più modulare prevede la suddivisione del sistema operativo in tre strati principali: user mode, kernel mode e hardware. Il meccanismo di TRAP funge da interfaccia tra questi livelli. Questo modello permette di caricare dinamicamente componenti aggiuntive come driver di dispositivi o file system senza dover riavviare o ricompilare l'intero sistema operativo.

Inoltre, i sistemi moderni consentono di utilizzare librerie condivise e DLL (Dynamic Link Libraries) per condividere codice tra applicazioni e ridurre la duplicazione del codice in memoria.

6 Altri Comandi

Alcuni comandi utili nei sistemi operativi includono:

- `man (comando)`: Per accedere al manuale dei comandi.
- `pwd`: Per visualizzare la cartella corrente.
- `ls`: Per elencare il contenuto della cartella.
- `cd`: Per cambiare la directory corrente.
- `chmod`: Per cambiare i diritti di accesso ai file.

Sistemi Operativi

Ionut Zbirciog

12 October 2023

1 Struttura di un sistema operativo

1.1 Monolitico

L'organizzazione stratificata del sistema operativo prevede una gerarchia a livelli (layers) costruiti uno sopra l'altro. Il sistema THE fu uno dei primi a implementare questa idea con sei livelli gerarchici:

1. Il livello 0 forniva una multiprogrammazione base della CPU.
2. Il livello 1 gestiva la memoria e allocava spazio per i processi.
3. Il livello 2 gestiva la comunicazione fra ogni processo e la console dell'operatore.
4. Il livello 3 gestiva i dispositivi di I/O.
5. Il livello 4 gestiva i programmi utente.
6. Il livello 5 gestiva l'operatore di sistema.

Un altro concetto di stratificazione era presente nel sistema MULTICS, descritto come una serie di anelli concentrici, dove quelli più interni avevano più privilegi di quelli esterni. Quando una procedura in un anello esterno voleva chiamare una procedura in un anello interno, doveva effettuare l'equivalente di una chiamata di sistema, ovvero un'istruzione di trap.

Vantaggi del kernel monolitico:

- Protezione delle risorse e dati critici.
- Separazione chiara dei compiti.

Svantaggi: Con l'evoluzione del sistema, questa struttura può diventare complessa e meno gestibile.
Proprietà di un kernel monolitico

1. Kernel Unificato: Tutte le funzionalità centralizzate in un unico kernel.
2. Interconnessione: Ogni componente ha la capacità di richiamare qualsiasi altro componente.
3. Scalabilità: Questa struttura può diventare complessa e meno gestibile con l'evoluzione del sistema.

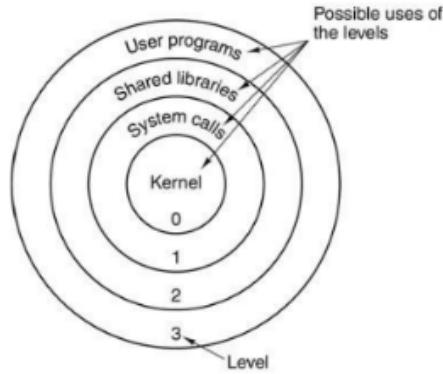


Figure 1: Struttura a strati MULTICS

1.2 Virtualizzazione

1.2.1 VM/370

Il cuore del sistema, monitor delle macchine virtuali, gira sul nudo hardware e realizza la multiprogrammazione fornendo non una ma tante macchine virtuali. Queste macchine virtuali sono la copia esatta dell'hardware, inclusi modalità kernel/utente, I/O, interruzioni e tutto ciò di cui dispone la macchina reale. Ogni macchina virtuale può eseguire un suo sistema operativo che viene eseguito direttamente sull'hardware.

Usi e vantaggi della virtualizzazione oggi:

- Hosting di server di posta, web, FTP sulla stessa macchina.
- Esecuzione di due o più sistemi operativi sulla stessa macchina (Windows e Linux).

Il monitor delle macchine virtuali è chiamato Hypervisor o VMM. Esistono tre tipi di Hypervisor:

1. Type 1: VMM viene eseguito sull'hardware (es. Xen).
2. Type 2: VMM ospitato nel sistema operativo (VirtualBox, VMWare).
 - (a) Hardware
 - (b) Sistema Operativo
 - (c) Applicativo di virtualizzazione
 - (d) Altro Sistema Operativo
 - (e) Vari applicativi
3. Hybrid: VMM all'interno del sistema operativo.

1.3 Container (Docker)

Oltre alla virtualizzazione completa, è possibile eseguire contemporaneamente più istanze di un sistema operativo su un'unica macchina, facendo sì che il sistema operativo stesso supporti sistemi diversi, o contenitori. I contenitori sono forniti dal sistema operativo host, come Linux o Windows, e generalmente eseguono solo la parte in modalità utente di un sistema operativo. Tutti i container condividono il kernel del sistema operativo host, tipicamente binari e librerie in modalità di sola lettura.

Vantaggi dei contenitori:

- Evita la duplicazione delle stesse risorse e file di sistema.
- Crea un ambiente protetto (Sand Box), aumentando la robustezza delle singole macchine.

Svantaggi:

- Non è possibile eseguire un sistema operativo completamente diverso dal sistema host.
- Non esiste un rigido partizionamento delle risorse come nelle macchine virtuali.

1.4 Exokernel e Unikernel

1.4.1 Exokernel

L'idea alla base dello schema exokernel è separare il controllo delle risorse dalla macchina estesa. L'exokernel è una versione semplificata dell'idea di container, in quanto non emula l'hardware, ma fornisce un kernel con solo alcune chiamate di sistema di cui ha bisogno l'applicativo. Il vantaggio dello schema exokernel è che risparmia uno livello di mappatura. Negli altri schemi ogni macchina virtuale pensa di avere un proprio SSD o disco, così il monitor delle macchine virtuali deve mantenere delle tabelle per rimappare gli indirizzi dei blocchi del disco. Con l'exokernel questa mappatura non è più necessaria in quanto deve solo tenere traccia di quale sia la macchina virtuale a cui è stata assegnata una certa risorsa. Tuttavia, nei container c'è più flessibilità in quanto è possibile installare ulteriori pacchetti/applicativi.

1.4.2 Unikernel

Il passo successivo è l'unikernel, che sono sistemi minimi progettati per gestire solo un'applicazione. Sono sistemi minimi basati su LibOS che contengono solo la funzionalità sufficiente a supportare un'unica applicazione, come un web server, su una macchina virtuale.

1.5 MicroKernel Client - Server

L'idea alla base del design a microkernel è ottenere un'alta stabilità suddividendo il sistema operativo in piccoli moduli ben definiti, uno dei quali è eseguito in modalità kernel e il resto come normali processi utenti separati. I processi di sistema comunicano attraverso il passaggio di messaggi, il quale è più lento di una chiamata di funzione come in un kernel monolitico. Le chiamate di sistema si basano sullo stesso meccanismo di messaggistica, implementato nel kernel minimale (Microkernel).

Sistemi Operativi

Ionut Zbirciog

19 October 2023

1 Processi

1.1 Definizione

Un processo è un programma in esecuzione.

In un sistema multiprogrammato, ciascuna CPU passa rapidamente da un processo all'altro, eseguendo ognuno per decine o magari centinaia di millisecondi. La CPU in realtà esegue un solo processo alla volta (la CPU viene assegnata a turni a diversi processi), ma nel corso di 1 secondo può elaborarne parecchi e dare l'illusione del parallelismo (**pseudoparallelismo**). Mentre si parla di **multiprocessore** quando ci sono 2 o più processori a livello hardware.

1.1.1 Il modello del processo

In questo modello tutto il software eseguibile sul computer, incluso il sistema operativo, è organizzato in un certo numero di processi. Un processo, che è un programma in esecuzione, include i valori attuali del contatore di programma, dei registri e delle variabili. Concettualmente, ogni processo ha la sua CPU, ma nella realtà la CPU passa da un processo all'altro, dando la sensazione all'utente di esecuzione in parallelo. Questo rapido passare avanti e indietro è detto multiprogrammazione.

1.1.2 Come funziona il processo

- Esecuzione sequenziale: Un unico program counter, ogni processo è in un'unica posizione, e la CPU, essendo una sola, passa da un processo all'altro in modo sequenziale.
- Esecuzione parallela: Ogni processo ha un proprio flusso di controllo (propri puntatori, propria area di memoria). Ogni volta che si passa da un processo all'altro, si salva il contatore di programma e le variabili nella tabella dei processi del primo processo e si ripristina il contatore di programma del secondo. All'interno del sistema operativo esiste un modulo (scheduler) che, in base alla priorità e a dei criteri detti policy di scheduling, decide quale processo deve essere assegnato prima o dopo alla CPU. Tutto questo avviene perché l'unico obiettivo del sistema operativo è di ottimizzare al massimo le risorse a disposizione, dando l'illusione di essere super veloce. Lo scheduler può prendere decisioni anche controllate, come per esempio scegliere un processo che sta per terminare.

In linea di principio, i processi multipli sono reciprocamente indipendenti, quindi hanno bisogno di mezzi esplicativi per dialogare tra loro. Il sistema operativo normalmente non offre garanzie di

tempestività o di ordine, per esempio, non offre la garanzia che ogni n secondi verrà assegnata la CPU ad ogni processo per t secondi. È un meccanismo molto complesso che può dipendere anche dal carico effettivo della macchina.

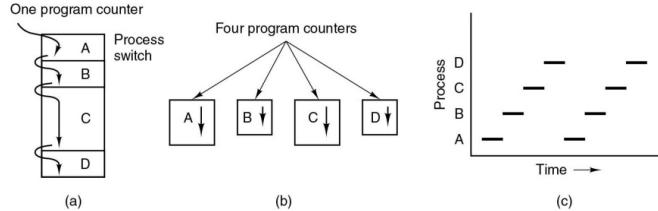


Figure 1: Esecuzione di processi: sequenziali e concorrenti

1.1.3 Informazioni associate a un processo

- PID (Process ID), UID (User ID), GID (Group ID)
- Spazio degli indirizzi di memoria
- Registri hardware (Program Counter)
- File aperti
- **Segnali**
- **Interrupt**

1.2 Gerarchia di Processi

Generalmente, i processi sono legati da una stretta gerarchia, ognuno è responsabile di altri processi perché devono sempre risalire a chi è responsabile, in quanto è l'unico modo per assicurarsi che una volta che uno termina, terminano tutti gli altri generati da quel processo. Avremo sempre la gerarchia Parent - Child. In UNIX, un processo speciale chiamato init (PID 1) è presente nell'immagine di boot. Quando comincia la sua esecuzione, esso legge un file che indica quanti terminali sono presenti. Poi esegue il fork di un nuovo processo per ogni terminale. Questi processi attendono che qualcuno esegua il login. Se il login avviene correttamente, il processo di login esegue una shell che accetta i comandi. Questi comandi possono far partire altri processi e così via. Nei moderni sistemi init, avvia kthreadd (PID 2), un processo per la gestione dei thread.

1.3 Creazione del Processo

La creazione di un processo può essere caratterizzata da 4 eventi principali:

1. Inizializzazione del sistema: All'avvio del sistema operativo vengono creati parecchi processi. Alcuni sono processi attivi, che interagiscono con gli utenti, alcuni sono processi in background (servizio per email, server web, ecc.), chiamati anche demoni.

2. Esecuzione di una chiamata di sistema per la creazione di un processo da parte di un processo in esecuzione (`fork()`).
3. Richiesta dell'utente di creare un nuovo processo (per esempio tramite bash).
4. Avvio di un lavoro in modalità batch (tramite script sh).

1.4 Termine di un Processo

Condizioni tipiche che terminano un processo:

1. Uscita normale (volontaria).
2. Uscita a causa di un errore (volontaria), ad esempio in C, ‘return 0’.
3. Errore ‘fatale’ (involontario), ad esempio in C, ‘segmentation fault’.
4. Ucciso da un altro processo (involontario).

Comandi per la gestione di un processo:

- ‘fork’: crea un processo, il figlio è un clone ”privato” del genitore e condivide alcune risorse del genitore (program counter, registri).
- ‘exec’: esegue un nuovo processo, viene utilizzato in combinazione con ‘fork’ (in C, ‘exec’ esegue comandi, quindi non eseguire mai comandi con ‘sudo’ in C).
- ‘exit’: causa la terminazione volontaria del processo, lo stato di uscita viene restituito al processo genitore.
- ‘kill’: invia un segnale a un processo (o a un gruppo), può causare la terminazione involontaria di un processo.

1.5 Gli Stati di un Processo

1. Running (In esecuzione): sta effettivamente utilizzando la CPU in quel momento.
2. Ready (Pronto): eseguibile, temporaneamente fermo per consentire l’esecuzione di un altro processo.
3. Blocked (Bloccato): non può essere eseguito fino a quando non si verifica un evento esterno (ad esempio, sta aspettando una risorsa). È uno stato fondamentale per evitare uno spreco dell’utilizzo della CPU a causa della lentezza del mondo fisico.

Dal punto di vista logico, il primo e il secondo stato sono simili. Il terzo stato è diverso dai primi due perché il processo non può essere eseguito, neanche se la CPU non ha niente da fare. Fra i 3 stati sono disponibili 4 transizioni.

1. Il processo si blocca in attesa di input, in alcuni sistemi il processo può eseguire una chiamata di sistema tipo `pause()`, per entrare in stato bloccato.
2. È causata dallo scheduler, infatti lo scheduler può decidere di finire l’esecuzione di un processo per sceglierne un altro, per esempio un processo con priorità maggiore.

3. E' causata dallo scheduler, infatti lo scheduler sceglie il processo che è in stato di pronto per eseguirlo.
4. Accade quando si verifica l'evento esterno di cui un processo era in attesa (come l'arrivo di un input). Se nessun altro processo è in esecuzione in quel momento, sarà innescata la transazione e il processo inizierà ad essere eseguito. Altrimenti dovrà attendere nello stato di pronto.

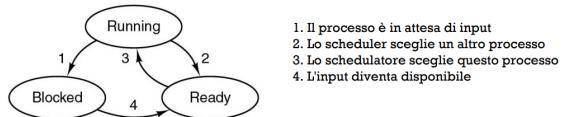


Figure 2: Gli stati di un processo

1.6 Signals vs Interrupts

Sono meccanismi utilizzati nei sistemi operativi e nelle applicazioni per gestire eventi asincroni (eventi indipendenti dal flusso di esecuzione del software).

1.6.1 Interrupts

- Origine: Dispositivi hardware (notificano l'avvenimento di eventi legati all'hardware, ad esempio quando si preme un tasto sulla tastiera).
- Gestione: Tramite routine di servizio di interrupt (ISR), routine specifiche e programmate all'interno del sistema operativo.
- Uso: Comunicazione tra hardware e software.
- Asincronia: Eventi che si verificano in modo asincrono e devono essere gestiti immediatamente.

Interrupt Vector: Associato a ciascun dispositivo di I/O e linea di interrupt, che contiene l'indirizzo dell' ISR (Interrupt Service Routine). Quando avviene un interrupt, per esempio dal disco, il program counter del processo, lo stato attuale del processo, e qualche registro vengono spinti sullo stack (viene salvato lo stato del processo). Il computer poi salta all'indirizzo specificato nel vettore di interrupt e viene eseguita la ISR rispettiva.

Tutti gli interrupt iniziano salvando i registri del processo attualmente in esecuzione. Questa procedura non può essere espressa in linguaggi come C, perciò sono eseguite da una piccola routine in assembly, generalmente la stessa per tutti gli interrupt.

Quando la routine è terminata, viene chiamata una procedura C per completare il lavoro per lo specifico tipo di interrupt. Quando ha finito il suo lavoro, magari rendendo pronto qualche processo viene chiamato lo scheduler per stabilire quale processo eseguire. A questo punto, il controllo viene restituito al codice in linguaggio assembly che carica i registri e la mappa della memoria del processo attuale e inizia ad eseguirlo.

Un processo può essere interrotto anche migliaia di volte durante la sua esecuzione, ma il concetto fondamentale è che dopo ogni interrupt il processo torna esattamente allo stato in cui si trovava prima che avvenisse l'interrupt.

1.6.2 Signals

- Origine: Eventi software (modo di comunicare tra i processi), generati da un processo o dal sistema operativo.
- Gestione: Gestori di segnali personalizzati o comportamento predefinito (ad esempio, CTRL + C per la terminazione di un processo o CTRL + Z per la sospensione di un processo).
- Uso: Gestione condizioni eccezionali nelle applicazioni.
- Asincronia: Inviati asincronamente ma possono essere gestiti in modo sincrono.

Un segnale, è un impulso asincrono trasmesso da un processo ad un altro, ed è uno degli strumenti di comunicazione tra processi. Tipicamente nessun dato viene trasmesso assieme al segnale. Ogni tipo di segnale ha un comportamento standard come SIGINT (interruzione del processo), SIGTERM (Terminazione del programma, iniziato dal comando kill se non diversamente specificato.), SIGKILL (Terminazione immediata (kill) del processo.). La gestione dei segnali può essere complicata, con alcune condizioni che sono specifiche per un thread, mentre altre no.

Ecco un esempio di gestione di un segnale in C:

```
void signalHandler(int signum){  
    printf("Interrupt signal %d received\n", signum);  
    exit(signum);  
}  
  
int main(){  
    signal(SIGINT, signalHandler); // Chiamata di sistema che imposta una funzione per gestire il segnale  
    while(1){  
        printf("Going to sleep....\n");  
        sleep(1);  
    }  
    return 0;  
}
```

2 Thread

Nei sistemi operativi tradizionali, ogni processo dispone di uno spazio degli indirizzi e di un singolo thread di controllo. Tuttavia, ci sono frequenti situazioni in cui è utile avere più thread di controllo in esecuzione nello stesso spazio di indirizzi, quasi in parallelo. La multithread execution implica che un processo può avere N thread in esecuzione.

Perché consentire più thread per processo? Innanzitutto, i thread sono dei miniprocessi leggeri, poiché a differenza dei processi, non hanno un loro PID, un loro spazio di indirizzamento, ecc. (lightweight processes). Inoltre, consentono un parallelismo efficiente in termini di spazio e di tempo, in quanto non devono essere gestiti dallo scheduler ma è il processo stesso che li deve gestire.

Un esempio per dimostrare l'utilità dei thread è la gestione delle pagine web da parte di un web server. Ipotizzando che ad ogni richiesta di un client, il web server avvia un nuovo processo per la gestione della richiesta, se ci fossero milioni di richieste, allora il web server dovrebbe avviare

milioni di processi simultanei, un numero molto elevato e che lo scheduler non sarebbe in grado di gestire. Invece di avviare un processo per ogni richiesta, il web server attribuisce un thread ad ogni richiesta, che è molto più leggero di un processo e che viene gestito direttamente dal web server e non dallo scheduler. Così facendo, il processo durante il tempo in cui è allocato alla CPU riesce a rispondere a N richieste.

I thread si trovano tutti nella stessa area di memoria del processo che li ha generati, quindi sono in grado di scrivere e leggere dalla stessa memoria condivisa. Questa caratteristica dei thread comporta anche dei problemi di sincronizzazione tra thread, poiché se un thread sta scrivendo su una variabile e un secondo thread legge e modifica la stessa variabile, il thread iniziale quando andrà a leggere sulla variabile si aspetterà di leggere quello che ha scritto lui, invece troverà qualcos'altro. Le uniche informazioni personali di ciascun thread sono: stack, registri, memoria, stato. Ciascun thread inoltre può chiamare qualsiasi chiamata di sistema supportata dal sistema operativo per conto del processo a cui appartiene.

I thread in POSIX:

1. `pthread_create`: crea un nuovo thread
2. `pthread_exit`: termina il thread chiamante
3. `pthread_join`: attende l'uscita di un specifico thread
4. `pthread_yield`: rilascia la CPU per consentire l'esecuzione di un altro thread
5. `pthread_attr_init`: crea e inizializza la struttura di attributi di un thread
6. `pthread_attr_destroy`: rimuove la struttura di attributi di un thread

2.1 Implementazione dei Thread nello Spazio Utente (JAVA)

Pro

- Sono gestiti dal kernel come processi a singolo thread.
- Possono essere eseguiti su sistemi operativi che non supportano direttamente i thread.
- Sono gestiti tramite una libreria.
- Offrono l'abilità di personalizzare l'algoritmo di scheduling per ogni processo e una maggiore scalabilità.

Contro

- Problemi con le chiamate di sistema bloccanti.
- Se un thread fa una chiamata che lo blocca, tutti gli altri thread nel processo vengono fermati.
- Gli errori di pagina, dove un programma accede a memoria non presente, possono bloccare l'intero processo quando sono causati da un thread a livello utente.
- I thread nello spazio utente non hanno un interrupt del clock, rendendo impossibile uno scheduling di tipo round-robin.
- Sebbene i thread a livello utente siano più veloci e flessibili, sono meno adatti per applicazioni in cui i thread si bloccano frequentemente, come i web server multithread.

2.2 Implementazione dei Thread nello Spazio Kernel

Pro

- Il kernel che gestisce i thread elimina la necessità di un sistema run-time per processo.
- Le chiamate di sistema che potrebbero bloccare un thread vengono implementate come chiamate di sistema.
- Alcuni sistemi riciclano i thread per ridurre i costi, invece che terminarli.
- La programmazione con thread richiede cautela per evitare errori.

Problemi aperti

- Molte procedure di libreria possono causare conflitti se un thread sovrascrive dati cruciali per un altro.
- L'implementazione di wrappers può evitare conflitti ma limita il parallelismo.
- La gestione dei segnali è complicata, con alcuni specifici per un thread e altri no.

Sistemi Operativi

Ionut Zbirciog

19 October 2023

Sincronizzazione e Comunicazione tra Processi

I processi hanno bisogno di un modo per comunicare in modo da condividere i dati e sincronizzarsi con altri processi durante l'esecuzione. Molto brevemente, i problemi sono 3. Il primo è relativo al modo in cui un processo può passare informazioni ad un altro. Il secondo è accettarsi che due o più processi o thread non si intralcino a vicenda. Il terzo riguarda la corretta sequenzialità quando vi sono delle dipendenze: se il thread A produce dei dati e il thread B li stampa, B deve attendere che A abbia prodotto qualche dato prima di iniziare a stampare.

Race Conditions

I processi possono lavorare insieme condividendo una parte di memorizzazione comune che ciascuno può leggere e scrivere. Questa memorizzazione condivisa può trovarsi nella memoria principale o può essere un file condiviso. Situazioni in cui due o più processi leggono o scrivono i medesimi dati condivisi e il risultato finale dipende dai tempi precisi in cui vengono eseguiti, sono chiamate *race conditions*. Ovvero, i processi "corrono" insieme per ottenere l'accesso ad una risorsa.

Regione Critica

Per evitare le race conditions serve una mutua esclusione, ossia un qualche sistema per essere certi che, se un processo sta usando un variabile o un file condiviso, agli altri processi venga impedito di fare la stessa cosa. La parte di programma in cui accede alla memoria condivisa è chiamata regione critica o sezione critica, quindi la soluzione alle race conditions è di non fare accedere due processi/thread contemporaneamente nella regione critica. Per ottenere una buona soluzione servono 4 condizioni da rispettare:

1. Due processi non possono trovarsi contemporaneamente all'interno delle rispettive regioni critiche
2. Non si possono fare ipotesi sulla velocità o sul numero di CPU
3. Nessun processo in esecuzione al di fuori della propria regione critica può bloccare altri processi
4. Nessun processo deve aspettare all'infinito per entrare nella propria regione critica

Mutua Esclusione con Busy Waiting

(a) Disabilitare gli Interrupt

Disabilitare gli interrupt di un processo appena entrato nella sua regione critica e li riabiliti quando ne esce. In questo modo non c'è la possibilità che un altro processo entri nella regione critica, poiché la riallocazione della CPU avviene ad ogni interrupt di clock e di altri interrupt. È una tecnica che può essere usata solo dal kernel per aggiornare alcuni dati critici, ma non è opportuno dare la possibilità ai processi utente di disabilitare gli interrupt. Funziona solo per sistemi a CPU singola (single core), poiché nei sistemi multicore, se gli interrupt di un core vengono disabilitati, gli altri core possono comunque interferire.

(b) Bloccare le Variabili

Proteggere le regioni critiche con variabili 0/1. Le 'corse' si verificano ora sulle variabili di blocco.

(c) Alternanza Stretta

La variabile *turn*, inizialmente a 0, tiene traccia dei turni dei processi che vogliono entrare nella regione critica. Il processo A vede che *turn* è 0, quindi entra nella regione critica impostando *turn* a 1. Nel mentre il processo B vede che *turn* è 1, quindi si mette in attesa in un rapido ciclo. L'azione di testare continuamente una variabile finché non è valorizzata si chiama *busy waiting*. Andrebbe generalmente evitato, dato che consuma tempo alla CPU. Di solito si usa quando l'attesa è relativamente breve.

- Processo A:

```
while(TRUE){  
    while(turn != 0);  
    critcal_region();  
    turn = 1;  
    noncritical_region();  
}
```

- Processo B:

```
while(TRUE){  
    while(turn != 1);  
    critcal_region();  
    turn = 0;  
    noncritical_region();  
}
```

Purtroppo, questa è un'altra non soluzione perché:
- Non permette ai processi di entrare nelle loro regioni critiche per due volte di seguito.
- Un processo fuori dalla regione critica può effettivamente bloccarne un altro violando la 3 condizione.

(d) Peterson's Algorithm

Alice e Bob vogliono usare un'unica postazione computer in un ufficio. Ma ci sono delle regole:

1. Solo una persona può usare il computer alla volta.
2. Se entrambi vogliono usarlo contemporaneamente, devono decidere chi va per primo.

Idea: Alice e Bob devono segnalare il loro interesse a usare il computer. Se l'altro non è interessato, la persona interessata può usarlo subito. Se entrambi mostrano interesse, registrano il loro nome su un foglio. Ma se scrivono quasi allo stesso tempo, l'ultimo nome sul foglio ha precedenza. Perché: garantisce che sempre uno avrà scritto e l'ultimo sarà quello che si prende la risorsa. La persona che non ha la precedenza aspetta finché l'altra ha finito. Una volta finito, la persona che ha usato il computer segnala che ha finito, e l'altra può iniziare.

```
#define N 2
```

```
int turn;
int interested[N];

void enter_region(int process){
    int other;
    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while(turn == process && interested[other] == TRUE);
}

void leave_region(int process){
    interested[process] = FALSE;
}
```

(e) TSL e XCHG

Istruzione TSL (Test and Set Lock)

È presente in computer con più processori. Legge il contenuto della memoria "lock", salva un valore non zero, e blocca altre CPU dall'accesso alla memoria. Purtroppo, anche disabilitando gli interrupt su un processore, non c'è garanzia che un processo "non faccia danni" da un'altra CPU e quindi, si bloccano tutti fino al termine dell'esecuzione di TSL. Viene fatto solo per operazioni atomiche, come per assegnare un valore.

Funzionamento: Quando *lock* è 0, un processo può impostare *lock* a 1 con TSL e accedere alla memoria condivisa. Al termine, il processo resetta *lock* a 0. Metodo per gestire Regioni Critiche:

- Processi chiamano *enter_region* prima di entrare nella regione critica e *leave_region* dopo.
- Se chiamati correttamente, garantisce la mutua esclusione.
- Se usati in modo errato, la mutua esclusione fallisce.

```

enter_region:
    TSL REGISTER, LOCK
    CMP REGISTER, #0
    JNE enter_region
    RET

leave_region:
    MOVE LOCK, #0
    RET

```

Istruzione XCHG: Scambia i contenuti di due posizioni atomicamente. Usata in tutte le CPU x86 Intel per sincronizzazione di basso livello.

Sleep e Wakeup

Nonostante l'algoritmo di Peterson funzioni, rimane il problema dello spinlock causato dal busy waiting, dove il processo tiene occupata la CPU in attesa di poter entrare nella sua regione critica.

La soluzione consiste nel permettere al processo in attesa di entrare nella sua regione critica di restituire volontariamente la CPU allo scheduler.

```

void sleep(){
    set own state to BLOCKED;
    give CPU to scheduler;
}

void wakeup(process){
    set state of process to READY;
    give CPU to scheduler;
}

```

Problema Produttore-Consumatore

Nel problema del produttore-consumatore, due processi condividono un buffer di dimensioni fisse. Il produttore inserisce informazioni nel buffer, mentre il consumatore le preleva.

```

#define N 100
int count = 0;

void producer(void){
    int item;
    while(TRUE){
        item = produce_item();
        if(count == N) sleep();
        insert_item(item);
        count++;
        if(count == 1) wakeup(cons);
    }
}

```

```

void consumer(void){
    int item;
    while(TRUE){
        if(count == 0) sleep();
        item = remove_item();
        count--;
        if(count == N - 1) wakeup(prod);
        consume_item(item);
    }
}

```

Problema: il produttore potrebbe svegliare il consumatore un attimo prima di andare in sleep, e nessuno lo risveglierebbe più perché non potendo più consumare, il produttore non sa se deve produrre o no.

Semafori

Fondamentalmente, il semaforo è una variabile che può essere 0 (nessun wakeup) o un valore positivo (wakeup in attesa). Su questa variabile si possono eseguire le seguenti operazioni:

- **down**: Se il valore del semaforo è maggiore di zero, questo valore viene decrementato, e il processo continua la sua esecuzione. Se il valore del semaforo è 0, il processo che ha invocato **down** viene bloccato e messo in una coda di attesa associata al semaforo (va a dormire, **sleep()**).
- **up**: Se il valore è 0, ci sono processi nella coda di attesa che vengono "svegliati" (eventualmente per entrare in competizione ed eseguire di nuovo **down**). In ogni caso, il valore viene incrementato e il processo continua la sua esecuzione.

Atomicità: Le operazioni sui semafori sono "indivisibili", evitando conflitti.

Problema Produttore-Consumatore

Utilizzo dei semafori per gestire l'accesso e la capacità di un buffer.

- **mutex** (mutual exclusion, accesso esclusivo).
- **full** (tutti i posti occupati).
- **empty** (tutti i posti liberi).

mutex previene eccessi simultanei, **full** e **empty** coordinano attività.

{6.2_produce_consumer_semaphore.c}

Problema Scrittori e Lettori

Regola: Ci sono tanti lettori (consumer) e un solo scrittore (producer). Ad esempio, si possono avere molteplici letture su un database, ma solo un singolo scrittore.

Funzionamento Sintetico:

- Il primo lettore blocca l'accesso al database.
- Lettori successivi incrementano un contatore.
- L'ultimo lettore libera l'accesso al database così lo scrittore può svolgere il suo lavoro.

{6.3_reader_writer_semaphore.c}

I semafori sono usati per sincronizzare più processi tra loro.

Mutex

Un *mutex* è una versione esplicita e semplificata dei semafori, usata per gestire la mutua esclusione di risorse o codice condiviso, quando non è necessario contare gli accessi e altri fenomeni. Può essere in due stati:

- **locked** (bloccato)
- **unlocked** (sbloccato)

Un solo bit può rappresentarlo, ma spesso viene utilizzato un intero (0 - unlocked, altri - locked). Due procedure principali sono `mutex_lock` e `mutex_unlock`:

- Quando un thread vuole accedere a una regione critica, chiama `mutex_lock`.
- Se il *mutex* è *unlocked*, il thread può entrare; se è *locked*, il thread attende.
- Al termine dell'accesso, il thread chiama `mutex_unlock` per liberare la risorsa.
- **IMPORTANTE:** Non si utilizza il *busy waiting*. Se un thread non può acquisire un lock, chiama `thread_yield` per cedere la CPU ad un altro thread.
- Alcuni pacchetti di thread offrono `mutex_trylock` che tenta di acquisire il lock o restituisce un errore senza bloccare. Questa opzione offre la possibilità di provare a prendere il lock, e se non è già occupato, continua ad eseguire altre operazioni senza sprecare tempo.

Mutexes in pthread

La libreria Posix Pthread fornisce funzioni per la sincronizzazione tra thread. Il *mutex* è una variabile che può essere *locked* o *unlocked* ed è utilizzato per proteggere le regioni critiche. Il thread tenta di bloccare (*lock*) un mutex per accedere alla regione critica. Se il mutex è *unlocked*, l'accesso è immediato e atomico. Se è *locked*, il thread attende.

- `pthread_mutex_init`: Crea il mutex.
- `pthread_mutex_destroy`: Distrugge il mutex.

- `pthread_mutex_lock`: Acquisisce il mutex o si blocca.
- `pthread_mutex_trylock`: Acquisisce il mutex o fallisce.
- `pthread_mutex_unlock`: Rilascia il mutex.
- `pthread_cond_init`: Crea una variabile condizionale.
- `pthread_cond_destroy`: Distrugge una variabile condizionale.
- `pthread_cond_wait`: Si blocca in attesa di un segnale.
- `pthread_cond_signal`: Segnala un altro thread e lo sveglia.
- `pthread_cond_broadcast`: Segnala multipli thread e li sveglia.

Semafori o Mutex?

Finalità

- **Mutex**: È utilizzato principalmente per garantire l'esclusione mutua. È destinato a proteggere l'accesso a una risorsa condivisa, garantendo che solo un thread possa accedervi alla volta.
- **Semaforo**: Può essere usato per controllare l'accesso a una risorsa condivisa, ma è anche spesso usato per la sincronizzazione tra thread.

Semantica

- **Mutex**: Di solito ha una semantica di "proprietà", il che significa che solo il thread che ha acquisito il mutex può rilasciarlo.
- **Semaforo**: Non ha una semantica di "proprietà". Qualsiasi thread può aumentare o diminuire il conteggio del semaforo, indipendentemente da chi lo ha modificato l'ultima volta.

Casistica

- **Per l'esclusione mutua**: Un mutex è generalmente preferibile. È più semplice (di solito ha operazioni di lock e unlock) e spesso offre una semantica più rigorosa e un comportamento più prevedibile.
- **Per la sincronizzazione tra thread**: Un semaforo può essere più adatto, specialmente quando si tratta di coordinare tra diversi thread o di gestire risorse con un numero limitato di istanze disponibili.

{6.4_produce_consumer_mutex.c}

NOTA

- **Protezione della risorsa condivisa:** `pthread_mutex_lock` assicura che solo un thread alla volta possa accedere e modificare la risorsa condivisa.
- **Attesa condizionale:** Quando un thread chiama `pthread_cond_wait`, due operazioni avvengono atomicamente. Il thread rilascia il mutex e mette il thread in uno stato di attesa sulla variabile condizionale. Anche se il produttore ha acquisito il mutex, non lo detiene mentre è in attesa sulla variabile condizionale. Questo permette al consumatore (o a un altro thread) di acquisire il mutex, fare le sue operazioni, e poi mandare un segnale alla variabile condizionale usando `pthread_cond_signal`.

Monitor

Un *monitor* raggruppa procedure, variabili e strutture dati. I processi possono chiamare le procedure di un monitor, ma non possono accedere direttamente alle sue strutture dati interne. Solo un processo può essere attivo in un monitor in un dato momento, garantendo la mutua esclusione. Il compilatore gestisce la mutua esclusione dei monitor, riducendo la probabilità di errori da parte del programmatore. Per gestire situazioni in cui i processi devono attendere, i monitor utilizzano variabili condizionali e due operazioni su di esse: `wait` e `signal`. A differenza dei semafori, le variabili condizionali non accumulano segnali; se un segnale viene inviato e non c'è un processo in attesa, il segnale viene perso. Linguaggi come Java supportano i monitor, permettendo una sincronizzazione e mutua esclusione più sicura e semplice in contesti multithreading. I metodi sono dichiarati `synchronized` in modo che solo un thread possa accedervi.

Scambio di Messaggi

Metodo di comunicazione tra processi usando due primitive: `send(dest, &msg)` e `receive(src, &msg)`. Può essere utilizzato in diversi scenari, compresi sistemi distribuiti. Problemi includono messaggi persi in rete, necessità di feedback per confermare la ricezione, gestione dei messaggi duplicati usando numeri sequenziali, autenticazione e denominazione dei processi. Malgrado l'inaffidabilità, lo scambio di messaggi è cruciale nello studio delle reti. Nel problema del produttore-consumatore, si può utilizzare una soluzione senza memoria condivisa, basata solo su messaggi. Questa soluzione impiega un totale di N messaggi, simili ai N posti del buffer nella memoria condivisa. Il consumatore invia al produttore N messaggi vuoti, e il produttore prende un messaggio vuoto, lo riempie e lo invia.

Barriere

Le barriere sono utilizzate per sincronizzare processi in fasi diverse. Quando un processo raggiunge una barriera, attende fino a quando tutti gli altri processi la raggiungono. Ad esempio, in calcoli paralleli su matrici, i processi non possono avanzare a un'iterazione successiva finché tutti non hanno terminato l'iterazione attuale. Sono utilizzate per sincronizzare processi in fasi diverse.

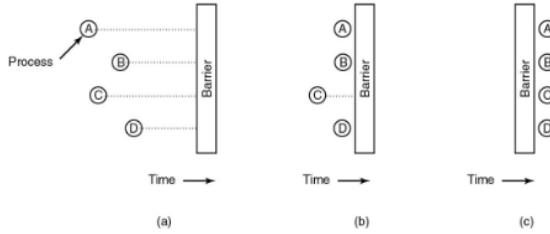


Figure 1: Barriere

Inversione delle Priorità

Dati tre thread T1, T2, T3 con rispettive priorità $p_1 > p_2 > p_3$, l'inversione delle priorità può verificarsi quando un thread con priorità più bassa detiene una risorsa che un thread con priorità più alta deve utilizzare. Soluzioni includono disattivare gli interrupt, *Priority Ceiling*, *Priority Inheritance*, e *Random Boosting*.

Read-Copy-Update

L'obiettivo del *Read-Copy-Update* è di accedere in modo concorrente senza lock, cercando di evitare l'incosistenza dei dati. L'idea è di aggiornare strutture dati consentendo letture simulate senza incappare in versioni inconsistenti dei dati. I lettori vedono o la versione vecchia o quella nuova, mai un mix delle due. È diffuso nel kernel dei sistemi operativi.

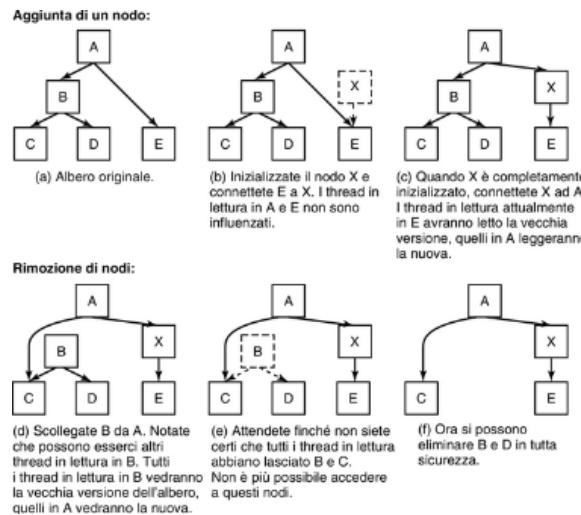


Figure 2: Inserimento e Cancellazione di un nodo

Sistemi Operativi

Ionut Zbirciog

2 November 2023

Introduzione allo Scheduling

In un sistema multiprogrammato molteplici processi e thread competono per la CPU, quindi è necessario scegliere a quale processo o thread assegnare la CPU. La parte del sistema operativo che effettua lo scheduling è detto *scheduler* e l'algoritmo che utilizza si chiama algoritmo di scheduling.

Molti problemi di scheduling per processi valgono anche per thread, nel caso in cui i thread sono gestiti dal kernel.

Nei sistemi batch storici, lo scheduling era lineare, ovvero i job venivano eseguiti in modo sequenziale. Con la multiprogrammazione, lo scheduling è diventato complessi a causa della concorrenza tra utenti.

Costi in termini di tempo del scheduling. Lo scambio di processi (*context switch*) è oneroso. - Cambio da modalità utente a modalità kernel. - Salvataggio dello stato del processo. - Esecuzione dell'algoritmo di scheduling. - Aggiornamento della MMU con la nuova mappa della memoria. - Potenziale invalidazione della memoria cache. Tutte queste operazioni possono consumare tempo alla CPU.

Problema di Scheduling dei Processi

Obiettivi:

- Equità - garantire un'equa condivisione della CPU a tutti i processi
- Imposizione della policy - garantire l'attuazione delle policy dichiarate
- Bilanciamento - mantenere tutti i componenti del sistema attivi

I processi alternano fasi di elaborazione CPU-intense con richieste di I/O. In genere un processo utilizza la CPU per un certo periodo senza interrompersi, poi fa una chiamata di sistema per leggere da un file o scrivere un file. Quando la chiamata di sistema è completa, riprende a usare la CPU fino a quando ha bisogno di leggere o di scrivere altri dati, e così via. Quando la CPU copia/preleva dati dalla RAM si parla di elaborazione e non di I/O.

Ci sono due tipologie di processi: 1. Processi *Compute-bound* (CPU-bound): Burst di CPU lunghi, attese di I/O infrequent, quindi passano molto tempo nella CPU. 2. Processi I/O-bound: Burst di CPU brevi, attese di I/O frequenti. Sono tali a causa della bassa necessità di calcoli, non della durata delle richieste di I/O.

Quando Effettuare lo Scheduling

- **Creazione Nuovo Processo:** Lo scheduler deve decidere quale processo tra figlio e padre scegliere per l'esecuzione. Essendo entrambi in stato di ready, si tratta di una normale decisione di scheduling, che può andare bene in entrambe le direzioni.
- **Uscita di un Processo:** Se un processo esce, occorre scegliere un altro dai processi pronti. Se nessuno è pronto, occorre eseguire un processo inattivo del sistema.
- **Blocco di un Processo:** Se un processo si blocca (I/O, semaforo, etc...), occorre selezionarne un altro.
- **Interrupt di I/O:** Una decisione di scheduling va presa quando si verifica un interrupt di I/O. Se l'interrupt proveniva da un dispositivo di I/O che adesso ha terminato il lavoro, qualche processo bloccato in attesa del dispositivo di I/O potrebbe ora essere pronto a partire. Sta allo scheduler stabilire se eseguire il processo che è appena diventato pronto, quello che era in esecuzione al momento dell'interrupt o qualche altro processo.

Tipologie di Algoritmi Scheduling e Prelazione

- **Non Preemptive (Senza Prelazione):** Sceglie un processo e lo lascia eseguire fino a quando si blocca o rilascia volontariamente la CPU. Anche se è eseguito per ore, non sarà sospeso forzatamente. In effetti durante gli interrupt del clock non vengono prese decisioni di scheduling. Dopo il completamento dell'elaborazione dell'interrupt del clock viene ripristinato il processo che era in esecuzione prima dell'interrupt, a meno che un processo di priorità più alta sia in attesa di un timeout ora soddisfatto.
- **Preemptive (Con Prelazione):** Sceglie un processo e lo lascia girare per un tempo massimo prefissato. Se alla fine dell'intervallo di tempo è ancora in esecuzione, il processo è sospeso e lo scheduler ne sceglie un altro da eseguire (se è disponibile). Uno scheduling con prelazione richiede che vi sia un interrupt del clock alla fine dell'intervallo per restituire il controllo della CPU allo scheduler. Se non è disponibile il clock, lo scheduling senza prelazione rimane l'unica possibilità. La prelazione non è rilevante solo per le applicazioni, ma anche per i kernel dei sistemi operativi, specie se monolitici. Oggi molti di essi hanno la prelazione; se non l'avessero, un driver male implementato o una chiamata di sistema molto lenta potrebbero intasare la CPU. In un kernel con prelazione, invece, lo scheduler può forzare uno scambio di contesto per il driver o la chiamata troppo lenti.

Scheduling nei Sistemi Batch

I sistemi batch sono ancora molto utilizzati in attività aziendali periodiche come elaborazione di paghe, inventari, ecc. Nei sistemi batch non ci sono utenti impazienti al loro terminale in attesa di una risposta rapida a una breve richiesta, quindi sono spesso accettabili algoritmi senza prelazione o algoritmi con prelazione con lunghi periodi per ciascun processo. Questo approccio riduce gli scambi di processo e migliora così le prestazioni. Gli algoritmi batch sono effettivamente abbastanza generali e spesso applicabili anche ad altre situazioni.

Obiettivi

- **Throughput:** massimizzare il numero di job per ora
- **Tempo di turnaround:** ridurre al minimo il tempo dallo start al termine di un job
- **Utilizzo della CPU:** mantenere la CPU sempre impegnata

Algoritmi di Scheduling nei Sistemi Batch

(a) First-Come, First-Served

È un algoritmo di scheduling senza prelazione. I processi vengono assegnati alla CPU nell'ordine in cui arrivano. Viene usata una singola coda di processi in stato di pronto, il primo job esegue immediatamente senza interruzioni. Quando il processo in esecuzione si blocca, viene eseguito il prossimo, quando il processo torna in stato di pronto, viene posto in fondo alla coda.

Vantaggi: È un algoritmo facile da capire e da implementare, basta una singola linked list. Inoltre, è equo in base all'ordine di arrivo dei processi; il processo da eseguire è posizionato in testa alla lista, gli altri processi sono aggiunti in coda in base all'ordine di arrivo.

Svantaggio: Può risultare in tempi di attesa molto lunghi per processi I/O-bound in presenza di un processo CPU-bound.

(b) Shortest Job First

È un algoritmo di scheduling senza prelazione. Si suppone che i tempi di esecuzione siano noti in anticipo, ad esempio dopo una previsione. Lo scheduler preleva il job più breve. Se eseguiti



Figure 1: Shortest Job First

nell'ordine A, B, C, D la media di esecuzione è di 14 min, mentre se eseguiti con SJF, nell'ordine B, C, D, A la media di esecuzione è di 11 min.

Vantaggio: L'algoritmo è ottimale nel minimizzare il tempo di turnaround medio quando tutti i job sono disponibili contemporaneamente.

Svantaggio: Se i job arrivano in momenti diversi, SJF potrebbe non essere ottimale.

(c) Shortest Remaining Time Next

È una versione con prelazione dell'algoritmo SJF. Con questo algoritmo, lo scheduler sceglie sempre il processo che impiegherà meno tempo per terminare l'esecuzione. Anche in questo caso, il tempo di esecuzione di ciascun processo deve essere noto in anticipo. All'arrivo di un nuovo job, il suo tempo totale è confrontato al tempo restante dei processi attuali. Se il nuovo job richiede meno tempo del processo attuale per terminare, il processo attuale viene sospeso ed è avviato il nuovo job.

Scheduling nei Sistemi Interattivi

In un ambiente con utenti interattivi, l'uso della prelazione è essenziale per evitare che un processo monopolizzi la CPU e neghi il servizio agli altri. Anche se intenzionalmente nessun processo viene eseguito all'infinito, un processo potrebbe escludere tutti gli altri a tempo indeterminato per un errore di programmazione. La prelazione è necessaria per prevenire questi comportamenti. Anche i server rientrano in questa categoria, dato che normalmente servono molti utenti (remoti), e tutti sempre di fretta.

Obiettivi

- **Tempo di risposta:** risposte rapide alle richieste dell'utente
- **Adeguatezza:** soddisfare le aspettative dell'utente in termini di tempi di risposta

Algoritmi di Scheduling nei Sistemi Interattivi

(a) Round-Robin Scheduling

È uno degli algoritmi di scheduling più vecchi, semplici, equi e utilizzati. Ogni processo riceve un intervallo di tempo detto "quanto" per l'esecuzione. Se il processo non ha terminato l'esecuzione, la CPU viene prelazionata per un altro processo. Se un processo si blocca, il passaggio avviene automaticamente. L'algoritmo è facile da implementare, infatti lo scheduler deve mantenere una lista di processi eseguibili, quando il processo esaurisce il suo quanto di tempo viene posto in fondo alla lista. **Problema:** L'unico problema del Round-Robin è la durata del "quanto". Supponendo



Figure 2: Round - Robin

un 1ms per il cambio di contesto e 4ms per il quanto, il 20% del tempo CPU è sprecato in overhead. Un quanto tra 20 e 50 ms è spesso ragionevole per bilanciare efficienza e reattività.

(b) Priority Scheduling

A ciascun processo è assegnata una priorità, e l'esecuzione è consentita al processo eseguibile con la priorità più alta. Per impedire che i processi siano eseguiti indefinitamente, lo scheduler può abbasare la priorità del processo in esecuzione ad ogni interrupt, se scende sotto una soglia, avviene lo scambio con un altro processo. Inoltre c'è la possibilità di assegnare un quanto di tempo per l'esecuzione del processo. Infine, è necessario avere un algoritmo che ne aumenti le priorità altrimenti, potrebbero finire a priorità 0.

Le priorità possono essere assegnate in modo statico, per esempio il costo computazionale in un data center, un utente che paga 50\$ ha una priorità diversa rispetto ad un utente che paga 100\$. Oppure possono essere assegnate in modo dinamico, basato sull'utilizzo della CPU o sul comportamento I/O-bound.

Spesso conviene raggruppare in processi in classi di priorità e usare lo scheduling a priorità tra le classi e all'interno delle classi usare Round-Robin.

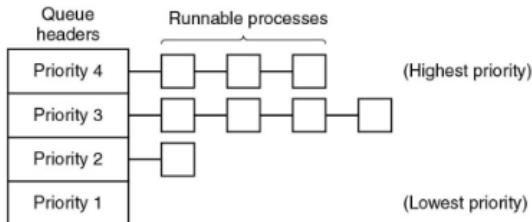


Figure 3: Priority Classes

Nell'immagine un sistema con 4 classi di priorità. Fintanto ci sono processi in priorità 4, si usa Round-Robin. Se vuota si passa alla 3 poi alla 2 e così via. È bene rivedere periodicamente le priorità, altrimenti i processi nelle classi di priorità inferiore rischiano di "morire d'inedia".

(c) Shortest Process Next

Nei sistemi batch, SJF produce sempre il minor tempo medio di risposta, quindi sarebbe comodo poterlo usare anche sui sistemi interattivi. La soluzione è fare stime basate sul comportamento passato (*Aging*). Supponiamo che il tempo stimato per comando per un certo terminale sia T_0 . Dopo una nuova esecuzione, T_1 , diventa $aT_0 + (1 - a)T_1$. Tramite la scelta di a , possiamo decidere se il processo di stima debba dimenticare in breve tempo le ultime esecuzioni o ricordarle a lungo. Con $a = \frac{1}{2}$ otteniamo:

$$T_0, \frac{T_0}{2} + \frac{T_1}{2}, \frac{T_0}{4} + \frac{T_1}{4} + \frac{T_2}{2}, \frac{T_0}{8} + \frac{T_1}{8} + \frac{T_2}{4} + \frac{T_3}{2},$$

dopo 3 esecuzioni il peso di T_0 è $1/8$. Questa tecnica è facile da applicare, specialmente con $a = \frac{1}{2}$.

(d) Guaranteed Scheduling

L'idea di base è fare promesse concrete sugli standard di prestazione e rispettarle. Se ci sono n utenti o processi, ciascuno ottiene $\sim \frac{1}{n}$ della potenza della CPU. Il sistema tiene traccia di quanta CPU ha ricevuto ogni processo dal momento della sua creazione (per esempio, 100s). Calcola quanto tempo CPU ogni processo dovrebbe avere: tempo creazione / n . Valuta il rapporto tra il tempo CPU consumato e quello dovuto. Esegue il processo con il rapporto più basso finché non supera il suo concorrente più vicino.

(e) Lottery Scheduling

L'idea di base è quella di dare ai processi un biglietto della lotteria per le diverse risorse del sistema, come il tempo della CPU. Ogni volta che si deve prendere una decisione di scheduling, viene estratto

un biglietto per decidere quale processo ottiene la risorsa. Ai processi più importanti possono essere assegnati dei biglietti extra, per aumentare le loro possibilità di vittoria.

Alcune proprietà:

- Reattività: Risponde velocemente ai nuovi processi grazie alla distribuzione dei biglietti.
- Cooperazione tra processi: i processi possono scambiarsi biglietti tra di loro.
- È adatto a situazioni dove altri metodi falliscono, per esempio un server video con diverse necessità di frequenze di fotogrammi.

(f) Fair-Share Scheduling

Tradizionalmente, ogni processo è oggetto di scheduling individualmente, senza considerare a chi appartenesse. Di conseguenza, se l'utente 1 ha 9 processi e l'utente 2 ne avvia 1, con il Round-Robin o le priorità uguali, l'utente 1 si prenderà il 90% della CPU e l'utente 2 solo il 10%. Per evitare questa situazione, alcuni sistemi considerano la proprietà di ciascun processo prima di considerarlo. Ogni utente riceve una frazione predefinita di CPU. Lo scheduler si assicura che ogni utente riceva la sua frazione, indipendentemente dal numero di processi posseduti.

Ad esempio, si consideri l'utente 1 con 4 processi A, B, C, D e l'utente 2 con 1 processo E. Una possibile sequenza con il Round-Robin è: A E B E C E D E..., mentre se l'utente 1 ha il doppio del tempo di CPU rispetto all'utente 2, otteniamo A B E C D E A B E...

Scheduling nei Sistemi Real-Time

In sistemi con vincoli real-time, la prelazione è, stranamente, non sempre necessaria, poiché i processi sanno che non possono essere eseguiti per lunghi periodi di tempo e generalmente fanno il loro lavoro e si bloccano rapidamente. Essi eseguono programmi per specifiche applicazioni, a differenza dei sistemi interattivi che possono eseguire programmi arbitrari.

Obiettivi:

- Rispetto delle scadenze - assicurarsi che i dati vengano elaborati nei tempi previsti
- Prevedibilità - assicurarsi che il funzionamento sia costante, specialmente in sistemi multi-mediali per evitare degradi della qualità

E' usato nei sistemi operativi in applicazioni in cui il tempo di risposta è fondamentale, per esempio in lettori musicali, monitoraggio in terapia intensiva, piloti automatici, controllo robotico in fabbriche, dove ritardi o mancati tempi di risposta possono avere gravi implicazioni.

Sono divisi generalmente in 2 categorie.

Hard Real-Time, nel caso di scadenze assolute da rispettare, **Soft Real-Time**, se una scadenza mancata di tanto in tanto è indesiderabile, ma c'è un certo grado di tollerabilità. In entrambi i casi il comportamento real-time si ottiene suddividendo il programma in un certo numero di processi, ognuno dei quali ha un comportamento prevedibile e noto in anticipo.

Gli eventi cui un sistema real-time può dover rispondere sono di tipo periodico, ovvero avvengono a intervalli regolari, oppure non periodico, ovvero avvengono in modo imprevedibile. Ad esempio,

se ci sono m eventi periodici, l'evento i avviene con un periodo P_i e richiede C_i secondi di tempo della CPU per gestire ogni evento, allora il carico può essere gestito solo se:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

Per esempio, consideriamo 3 eventi periodici con periodi di 100ms, 200ms, 500ms e tempi richiesti per la CPU di 50ms, 30ms, 100ms. Allora abbiamo $0.5 + 0.15 + 0.2 \leq 1$, quindi il sistema è schedulabile.

Gli algoritmi di scheduling real-time possono essere statici o dinamici. I primi prendono le loro decisioni di scheduling prima che il sistema inizi l'esecuzione, i secondi durante l'esecuzione. Lo scheduling statico funziona solo dove è disponibile in anticipo un'informazione perfetta riguardo al lavoro da svolgere e le scadenze da rispettare. Gli algoritmi di scheduling dinamico non hanno queste restrizioni.

Scheduling di Thread

Un processo può avere molti processi figli sotto il suo controllo. È del tutto possibile che il processo principale abbia un'idea eccellente di quale dei suoi figli sia il più importante e quale meno. Finora, gli algoritmi di scheduling non accettano input dai processi utenti riguardo alle decisioni di scheduling, spesso portando a decisioni sub-ottimali. La soluzione a questo problema è di dividere il meccanismo di scheduling dalla politica di scheduling, ciò significa che l'algoritmo di scheduling è in qualche modo parametrizzato, ma i parametri possono essere forniti dai processi utente. Per esempio, supponiamo un kernel con algoritmo di scheduling a priorità. Il kernel mette a disposizione una chiamata di sistema che permette a un processo di impostare le priorità dei suoi figli. In questo modo, il genitore può influenzare lo scheduling dei suoi figli senza controllarlo direttamente. In conclusione, il meccanismo sta nel kernel, mentre le policy sono definite dal processo utente.

Lo scheduling differisce in base al tipo di thread, se sono thread a livello utente o a livello kernel.

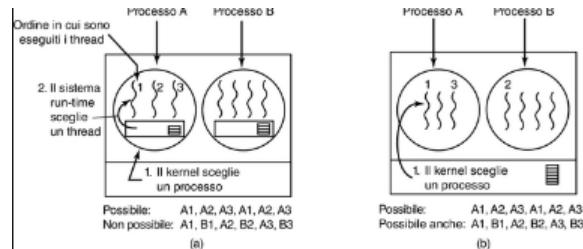


Figure 4: User Threads and Kernel Threads

- **Thread a livello utente:** il kernel ignora l'esistenza dei thread; sceglie un processo per il suo quanto. Lo scheduler interno al processo decide quale thread eseguire senza interruzioni del clock. Questo porta ad avere thread che possono consumare l'intero quanto del processo, influenzando solo il processo interno e non gli altri. Lo scambio dei thread avviene con poche istruzioni, dato che ci pensa il processo al suo interno. Un thread utente, se richiede I/O, sospende l'intero processo.

- **Thread a livello kernel:** il kernel seleziona un thread specifico per l'esecuzione; se un thread eccede il quanto, viene sospeso. Lo scambio del thread comporta uno scambio di contesto, quindi è più lento. Un thread kernel, se richiede I/O, non sospende tutto il processo ma solo il thread stesso.

Sistemi Operativi

Ionut Zbirciog

14 November 2023

Gestione della memoria nei sistemi operativi

La memoria principale (RAM) è una risorsa fondamentale che va gestita attentamente, anche se cresce rapidamente, i programmi crescono più velocemente. Il desiderio è quello di avere una memoria privata, grande, veloce e persistente (non volatile), ma la tecnologia di oggi ancora non permette di avere questo tipo di memoria. Nel corso degli anni si è sviluppato il concetto di gerarchia della memoria, i computer possono avere pochi megabyte di memoria molto veloce e molto costosa, qualche gigabyte di memoria abbastanza veloce ma meno costosa e volatile, e qualche terabyte di memoria lenta, poco costosa e non volatile. È compito del sistema operativo astrarre questa gerarchia. La parte del sistema operativo che gestisce la gerarchia della memoria è detto **Gestore della Memoria** i cui compiti sono: tener traccia della memoria in uso, allocare nuova memoria, e deallocare memoria

MEMORY ABSTRACTION

Memoria senza Astrazione

Il modo più semplice per astrarre la memoria è non farlo. Quindi il processo va ad utilizzare direttamente la memoria fisica. Questo tipo di "astrazione" funziona quando sulla macchina viene eseguito un solo programma. Questo modello ha fallito quando erano più di un programma a essere eseguiti sulla macchina. Inoltre, bastava far accedere il processo all'area di memoria del sistema operativo per causare danni.

Monoprogrammazione

Anche con il modello della memoria fisica, esistono tre sottomodelli di organizzazione della memoria:

1. Il sistema operativo può trovarsi in fondo alla RAM, usato sui mainframe e sui minicomputer.
2. Il sistema operativo si trova in ROM, usato nei sistemi embedded, incannato nella memoria ROM
3. Il sistema operativo e i drivers in RAM + ROM, usato sui primi modelli di personal computer, dove la parte del sistema nella ROM è chiamata BIOS (Basic Input Output System).

Multiprogrammazione

È però possibile eseguire più programmi contemporaneamente anche senza astrazione della memoria; il sistema operativo deve salvare l'intero contenuto della memoria in un file su memoria non volatile, quindi prelevare ed eseguire il programma successivo. Finché in memoria c'è un solo programma per volta, non ci sono conflitti. Questo concetto (swapping) verrà approfondito più avanti.

Un approccio che si ha avuto è il cosiddetto Naive Approach, ovvero caricamento di più programmi in memoria fisica consecutivamente, senza astrazione dell'indirizzo. Per esempio: Avendo due programmi da 16Kb, il programma A (0 - 16380) inizia con l'istruzione JMP 24, il programma B (16384 - 32764) inizia con l'istruzione JMP 28. Se sono caricati in sequenza, l'istruzione JMP 28 punta all'area di memoria 28, che appartiene al programma A, quindi punta ad uno spazio di memoria sbagliato, causando conflitti durante l'esecuzione e crash dei programmi.

Astrazione della Memoria

REGISTRI BASE E LIMITE

Per permettere a più applicazioni di risiedere in memoria contemporaneamente senza interferire l'una con l'altra devono essere risolti due problemi: protezione e rilocazione. Per quanto riguarda la protezione, abbiamo una soluzione primitiva usata sull'IBM 360: etichettare pezzi di memoria con una chiave di protezione e confrontare la chiave del processo in esecuzione con quella di ogni parola di memoria prelevata.

Una soluzione migliore è inventare una nuova astrazione della memoria, lo spazio degli indirizzi. Uno spazio degli indirizzi è l'insieme degli indirizzi che un processo può usare per indirizzare la memoria. Ogni processo ha il suo spazio degli indirizzi personale, indipendente da quelli degli altri processi. Più difficile è capire come dare a ogni programma il suo spazio degli indirizzi. L'indirizzo 28 di un programma è una locazione fisica in memoria diversa dall'indirizzo 28 di un altro programma.

Implementazione con registri base e limite, sono due registri presenti in molte CPU. Il registro base contiene l'indirizzo fisico di inizio di un programma in memoria. Il registro limite contiene la lunghezza del programma.

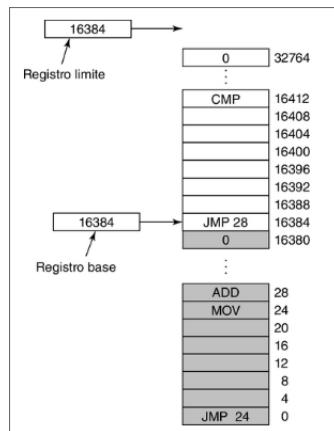


Figure 1: Registro Limite e Registro Base.

Ogni volta che un processo fa riferimento alla memoria, per prelevare un'istruzione o per leggere o scrivere una parola dati, prima di inviare l'indirizzo sul bus di memoria l'hardware della CPU aggiunge automaticamente il valore di base all'indirizzo generato tramite il processo. Contemporaneamente controlla se l'indirizzo offerto sia uguale o maggiore del valore nel registro limite, nel qual caso è generato un errore e l'accesso viene terminato.

Vantaggi: Offre a ciascun processo uno spazio di indirizzi protetto e separato

Svantaggi: Necessità di eseguire somme e confronti ad ogni accesso alla memoria, il che può essere lento.

SWAPPING

La strategia più semplice, chiamata swapping (scambio) dei processi, consiste nel prelevare ciascun processo nella sua totalità, eseguirlo per un certo tempo, quindi porlo nuovamente nella memoria non volatile. I processi inattivi vengono archiviati per la maggior parte su memoria non volatile, così non occupano memoria mentre non sono in esecuzione.

L'altra strategia, chiamata memoria virtuale, consente ai programmi di essere eseguiti anche quando sono solo parzialmente nella memoria principale.

Quando lo swapping crea più spazi vuoti nella memoria (frammentazione della memoria), è possibile combinarli tutti in un unico spazio vuoto spostando tutti i processi il più in basso possibile. Questa tecnica è conosciuta come memory compaction, di solito non viene attuata perché richiede troppo tempo alla CPU.

Un punto che vale la pena sottolineare è quanta memoria dovrebbe essere allocata per un processo quando viene creato o quando viene riportato in memoria dal disco mediante lo swapping. Se i processi sono creati con una dimensione fissa che non cambia mai, l'allocazione è semplice: il sistema operativo alloca esattamente il necessario, né più né meno.

Se invece i segmenti dei dati dei processi possono crescere, appena il processo prova a crescere sorge un problema.

Una soluzione a questo problema è di allocare memoria extra durante lo swapping o lo spostamento dei processi. Se un processo non può crescere nella memoria e l'area di swap del disco o dell'SSD è piena, il processo deve essere sospeso finché non sia liberato dello spazio (oppure terminato).

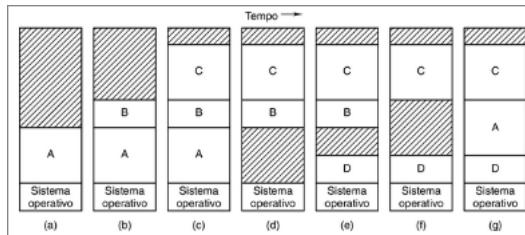


Figure 2: Funzionamento di un sistema di swapping.

Il funzionamento di un sistema di swapping è illustrato nella figura sopra. All'inizio solo il processo A è in memoria. Poi i processi B e C vengono creati o prelevati dalla memoria non volatile. Successivamente viene attuato lo swapping su memoria non volatile di A. Quindi arriva D ed esce B. Alla fine A ritorna, e poiché ora è in una posizione diversa, i suoi indirizzi devono

essere rilocati dal software al momento dello swapping o (più probabilmente) dall'hardware durante l'esecuzione del programma. In questo caso i registri base e limite così funzionerebbero bene.

GESTIONE DELLA MEMORIA LIBERA

Quando la memoria è assegnata dinamicamente, il sistema operativo deve gestirla. In termini generali, ci sono due modi per tener traccia dell'utilizzo della memoria: bitmap e liste.

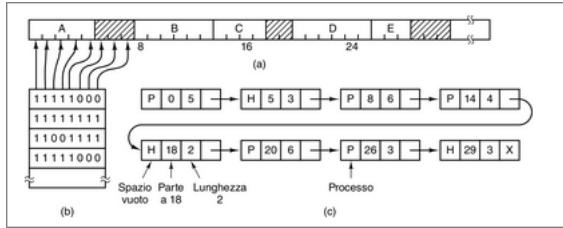


Figure 3: Bitmap e Liste nella gestione della memoria.

BITMAP

La memoria, con una bitmap, è divisa in unità di allocazione che possono essere piccole come qualche parola o grandi come vari kilobyte. A ogni unità di allocazione corrisponde un bit della bitmap, che è 0 se l'unità è libera e 1 se è utilizzata (o viceversa). La dimensione della bitmap dipende solo dalla dimensione della memoria e dalla dimensione dell'unità di allocazione, quindi una bitmap fornisce un modo semplice per tener traccia delle parole di memoria in una quantità fissa di memoria. Il problema principale è che, se si stabilisce di portare un processo di k unità in memoria, il gestore della memoria deve cercare nella bitmap per trovare una serie di k bit 0 consecutivi nella mappa. Cercare in una bitmap una serie di una certa lunghezza è un'operazione lenta.

LISTE

Un altro sistema per tenere traccia della memoria è mantenere una lista concatenata di segmenti di memoria allocati e liberi, in cui un segmento contiene un processo oppure è uno spazio vuoto fra due processi. Ogni voce della lista specifica uno spazio vuoto (H) o un processo (P), l'indirizzo da cui parte, la lunghezza e il puntatore alla voce successiva. Nella realtà viene spesso usata una doppia linked list, poiché rende più facile gestire lo spazio libero. Può controllare facilmente se il precedente spazio è libero. Può regolare facilmente i puntatori.

SCHEMI DI ALLOCAZIONE DELLA MEMORIA

Quando processi e spazi vuoti sono tenuti su una lista ordinata per indirizzo, è possibile usare vari algoritmi per allocare la memoria per un processo creato (o già esistente e scambiato da disco o SSD).

1. **First Fit:** Il gestore della memoria scorre la lista dei segmenti finché non trova uno spazio vuoto abbastanza grande. Lo spazio viene suddiviso in due parti, una per il processo e l'altra

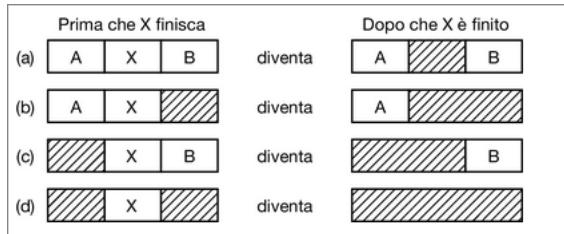


Figure 4: Struttura di una lista nella gestione della memoria.

per la memoria inutilizzata, tranne nel caso statisticamente improbabile che lo spazio coincida perfettamente. First fit cerca il meno possibile, quindi è un algoritmo veloce.

2. **Next Fit:** È una variazione di first fit; opera allo stesso modo, ma tiene traccia dei luoghi nei quali ha trovato uno spazio adatto. La volta seguente che viene chiamato per cercare uno spazio, cerca nella lista a partire dal punto dove era rimasto l'ultima volta, invece di partire dall'inizio come fa sempre first fit. Più lento di First Fit.
3. **Best Fit:** Cerca all'interno della lista dall'inizio alla fine, prendendo lo spazio più piccolo che sia comunque adatto. Anziché occupare solo in parte uno spazio grande che potrebbe essere necessario in seguito, best fit prova a cercare lo spazio della dimensione più vicina a quanto richiesto, per abbinare al meglio le richieste e gli spazi disponibili.
4. **Worst Fit:** Si prende sempre lo spazio disponibile più grande e lascia uno spazio di risulta abbastanza grande da essere utile.
5. **Quick Fit:** Mantiene liste divise per alcune delle dimensioni richieste più comunemente. Per esempio potrebbe esserci una tabella con n voci, di cui la prima è un puntatore alla testa di una lista di spazi di 4 KB, la seconda è un puntatore alla lista degli spazi da 8 KB, la terza a quelli da 12 KB e così via. Gli spazi da 21 KB potrebbero essere insieme a quelli da 20 KB oppure in una lista speciale degli spazi "strani". Con quick fit la ricerca di uno spazio della dimensione richiesta è molto veloce, ma presenta lo stesso svantaggio di tutti gli schemi ordinati per dimensione: quando un processo termina o viene scambiato su disco, trovare i suoi vicini per vedere se sia possibile un'unione è dispendioso. Se non viene eseguita l'unione, la memoria si frammenta rapidamente in un gran numero di piccoli spazi non adatti ad alcun processo.
6. **Buddy Allocation:** Migliora la performance di coalescenza del Quick Fit

BUDDY ALLOCATION

Funzionamento: La memoria inizia come un singolo pezzo contiguo. Ad ogni richiesta, la memoria viene divisa secondo una potenza di 2. I blocchi di memoria contigui vengono uniti quando rilasciati.

Questo algoritmo porta a una considerevole frammentazione interna, perché se è necessario un pezzo costituito da 65 pagine si dovrà chiedere e ottenere un pezzo da 128 pagine.

Per risolvere questo problema, Linux dispone di un secondo allocatore di memoria, l'allocatore a slab, che prende i pezzi usando l'algoritmo buddy ma poi da questi ritaglia gli slab (unità più piccole)

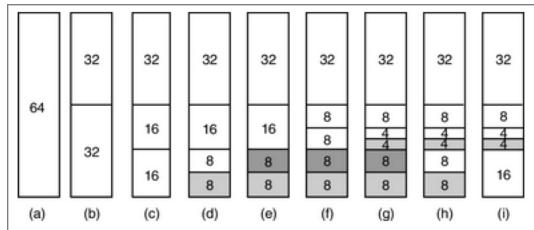


Figure 5: Esempio di funzionamento del buddy allocation.

e li gestisce separatamente. Lo slab allocator è usato per gestire l'allocazione di memoria in heap, di oggetti come array, linked list, struct, ecc... Il kernel spesso ha bisogno di creare e distruggere piccoli oggetti di dimensioni e tipi specifici. Senza ottimizzazione, questa operazione potrebbe portare a una significativa frammentazione della memoria. Nello slab allocation, la memoria è divisa in blocchi chiamati "slabs". Questi slab vengono ulteriormente suddivisi in chunk di dimensioni uniformi, adeguati per ospitare un oggetto di un certo tipo. Un slab può essere in uno dei seguenti stati: pieno (tutti i chunk pieni), parzialmente pieno (alcuni chunk sono pieni) o vuoto (tutti i chunk sono vuoti.). Quando un oggetto viene deallocated, non viene immediatamente restituito al sistema come memoria libera. Viene mantenuto nella cache in modo che, se viene richiesta un'altra istanza dello stesso tipo di oggetto, possa essere rapidamente riallocato senza l'overhead di inizializzazione.

Sistemi Operativi

Ionut Zbirciog

14 November 2023

ALGORITMI DI SOSTITUZIONE DELLE PAGINE

Quando si verifica un page fault, per far spazio alla pagina entrante il sistema operativo deve scegliere una pagina da sfrattare (rimuovere dalla memoria). Se la pagina da rimuovere è stata modificata mentre era in memoria, deve essere riscritta sulla memoria non volatile per aggiornare la copia su disco o SSD. Se invece la pagina non è stata modificata, la copia su disco o SSD è già aggiornata e non c'è bisogno di riscrivere. La pagina da leggere sovrascrive semplicemente la pagina sfrattata.

Sarebbe possibile prendere una pagina a caso da rimuovere a ogni page fault, ma le prestazioni del sistema migliorano molto se si sceglie una pagina non particolarmente utilizzata. Se si sfrattasse una pagina usata frequentemente, con ogni probabilità dovrebbe essere riportata in memoria a breve, con il risultato di un ulteriore sovraccarico.

1. L'algoritmo ottimale di sostituzione delle pagine

Ciascuna pagina può essere etichettata con il numero di istruzioni da eseguire prima di ricevere un riferimento per la prima volta. L'algoritmo di sostituzione ottimale indica che deve essere rimossa la pagina con l'etichetta con il numero più alto. Se una pagina non sarà usata per 8 milioni di istruzioni e un'altra pagina per 6 milioni di istruzioni, rimuovere la prima allontana il page fault che la ricaricherebbe il più in là possibile.

Il solo problema di questo algoritmo è che è irrealizzabile. Al momento del page fault, il sistema operativo non ha modo di sapere in quale momento verrà fatto il prossimo riferimento a ciascuna delle pagine. Tuttavia, eseguendo un programma su un simulatore e tenendo traccia di tutti i riferimenti alle pagine, è possibile implementare una sostituzione delle pagine ottimale sulla seconda esecuzione, usando le informazioni sui riferimenti alle pagine raccolte durante la prima esecuzione. In questo modo è possibile confrontare le prestazioni degli algoritmi realizzabili con quelle del migliore algoritmo possibile.

2. Not Recently Used (NRU)

Al fine di consentire al sistema operativo la raccolta di statistiche utili sull'uso delle pagine, molti computer con memoria virtuale hanno due bit di stato, R e M, associati a ciascuna pagina. R viene impostato quando si fa riferimento alla pagina (in lettura o scrittura). M è impostato quando la pagina viene scritta (cioè modificata). I bit sono contenuti in ciascuna voce della tabella delle pagine. I bit vengono aggiornati dall'hardware ad ogni accesso.

I bit R e M possono essere usati per costruire un algoritmo di paginazione semplice come il seguente. All'avvio di un processo, entrambi i bit di pagina di tutte le pagine sono impostati a 0 dal sistema operativo. Periodicamente (per esempio a ogni interrupt del clock), il bit R è ripulito, per contraddistinguere le pagine che non hanno avuto riferimenti recentemente da quelle che ne hanno avuti. Quando avviene un page fault, il sistema operativo ispeziona tutte le pagine e le divide in 4 categorie basate sui valori attuali dei loro bit R e M:

- Classe 0: nessun riferimento, non modificata.
- Classe 1: nessun riferimento, modificata.
- Classe 2: riferimento, non modificata.
- Classe 3: riferimento, modificata.

Le pagine di classe 1 sembrano a prima vista impossibili, ma appaiono quando un interrupt del clock azzerà il bit R di una pagina di classe 3. Gli interrupt del clock non azzerano il bit M perché questa informazione è necessaria per sapere se la pagina deve essere riscritta su disco o meno. Azzerare R ma non M produce una pagina di classe 1. In altre parole, una pagina di classe 1 è stata modificata molto tempo fa e da allora non è stata più toccata.

L'algoritmo NRU (Not Recently Used) rimuove una pagina a caso dalla classe non vuota con il numero più basso. I vantaggi sono: Semplicità, efficienza implementativa e prestazioni accettabili.

3. First-In, First-Out (FIFO)

Descrizione FIFO è un algoritmo di paginazione che elimina la pagina più vecchia in memoria.

Implementazione Il sistema operativo rimuove la pagina in testa alla lista (la più vecchia) durante un page fault, aggiungendo la nuova pagina in coda.

Problema Nel contesto informatico, la pagina più vecchia potrebbe ancora essere frequentemente utilizzata (ricorsione), rendendo FIFO poco efficace.

Conclusione A causa di queste limitazioni, FIFO è raramente utilizzato nella sua forma più semplice.

4. Seconda Chance

Una semplice modifica a FIFO che evita il problema di gettare una pagina usata di frequente consiste nel controllare il bit R della pagina più vecchia. Se è 0, la pagina è vecchia e inutilizzata e viene così sostituita immediatamente. Se R è 1, il bit viene azzerato, la pagina è posta in fondo all'elenco e il momento in cui è stata caricata in memoria viene aggiornato per farla sembrare appena arrivata. Poi la ricerca continua. L'operatività di questo algoritmo, detto seconda chance, è illustrata nella figura.

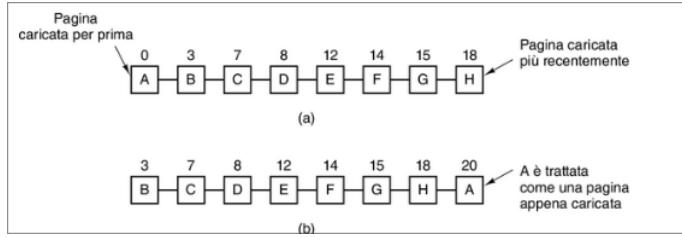


Figure 1: Seconda Chance.

Supponiamo che si verifichi un page fault all’istante 20.

La pagina più vecchia è A, arrivata al momento 0, all’inizio del processo. Se A ha il bit R azzerato, allora è rimossa dalla memoria, sia scrivendola su memoria non volatile (se è sporca) sia semplicemente scaricandola (se è pulita). Se invece il bit R è impostato, A viene portata in fondo alla lista e il suo “momento di caricamento” è reimpostato all’attuale (20). Anche il bit R è azzerato. La ricerca della pagina adatta prosegue con B.

Quello che la seconda chance sta cercando è una vecchia pagina che non sia stata oggetto di riferimenti durante l’ultimo intervallo del clock. Se tutte le pagine hanno avuto riferimenti, la seconda chance degenera in un FIFO puro.

5. Clock

Funzionamento

Lista circolare dei frame di pagina con un puntatore simile ad una lancetta di un orologio per identificare la pagina più vecchia. Quando avviene un page fault, la pagina indicata dalla lancetta viene controllata. Se il suo bit R è 0 viene sfrattata, la nuova pagina viene inserita al suo posto nell’orologio e la lancetta viene spostata in avanti di una posizione. Se R è 1, viene azzerato e la lancetta passa alla pagina successiva. Questo processo è ripetuto finché viene trovata una pagina con R = 0. È più efficiente rispetto a Seconda chance e FIFO.

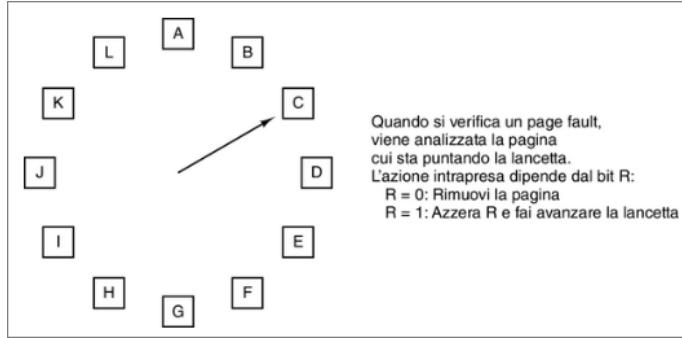


Figure 2: Clock.

6. Least Recently Used (LRU)

L'idea del algoritmo è di sfrattare la pagina rimasta inutilizzata per più tempo. Teoricamente l'algoritmo è fattibile, il problema è l'implementazione. Per implementare tale algoritmo è necessario tenere in memoria una lista concatenata di tutte le pagine, con quelle più usate in testa e quelle meno usate in coda. La difficoltà sta nel fatto che l'elenco deve essere aggiornato a ogni riferimento alla memoria. Trovare una pagina in memoria, cancellarla e poi portarla in testa è un'operazione che costa del tempo, anche se eseguita nell'hardware.

Esistono però altri metodi per implementare l'LRU con hardware speciale. Usare un contatore a 64 bit per ogni riferimento a memoria. Alla generazione di un page fault, si rimuove la pagina con contatore più basso, indicando l'uso meno recente.

7. Simulazione del LRU via software - algoritmo NFU (Not Frequently Used)

Associa ad ogni pagina un contatore, incrementato con ogni interrupt del clock in base al bit R. I contatori tengono traccia del numero di riferimenti di ciascuna pagina. Quando avviene un page fault, la scelta di quale pagina sostituire cade su quella con il contatore più basso.

Il problema principale dell'NFU è che non dimentica mai nulla. Per esempio, nel caso di un compilatore multipass (a molteplici passaggi), le pagine usate frequentemente durante il passaggio 1 avrebbero un conteggio alto anche nei passaggi successivi. Infatti, se accade che il passaggio 1 è quello con il tempo di esecuzione più alto fra tutti i passaggi, le pagine contenenti il codice per i passaggi seguenti potrebbero avere sempre conteggi inferiori alle pagine del passaggio 1, quindi il sistema operativo rimuoverebbe pagine utili invece di pagine non più utilizzate.

Il NFU si può migliorare semplicemente con il concetto di aging. Si usa un numero limitato di bit (8), ad ogni interrupt del clock i bit vengono spostati a destra, ma prima dello shifting, il bit R viene aggiunto a sinistra. Quando avviene un page fault, viene rimossa la pagina con il contatore più basso. È evidente che una pagina che non abbia riferimenti, diciamo, per quattro cicli del clock, avrà 4 zero consecutivi nel suo contatore e avrà così un valore inferiore rispetto a una senza riferimenti per tre cicli del clock.

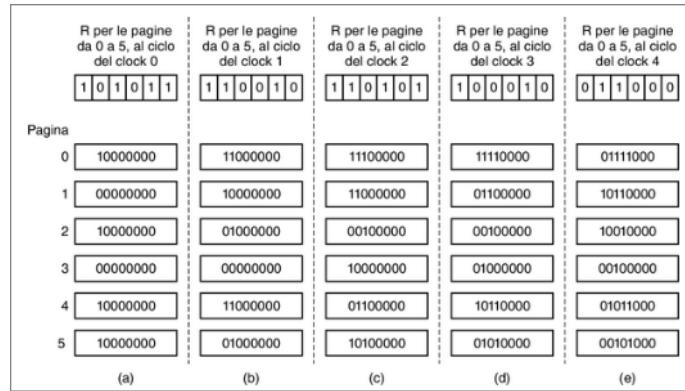


Figure 3: NFU con aging.

Consideriamo come esempio la pagina 1.

- a. Non è modificata ed ha valore 0 0 0 0 0 0 0.
- b. Viene modificata ed ha valore 1 0 0 0 0 0 0.
- c. Viene modificata ed ha valore 1 1 0 0 0 0 0.
- d. Non viene modificata ed ha valore 0 1 1 0 0 0 0.

Adesso consideriamo le pagine 3 e 5.

- c. Entrambe hanno avuto accesso:

- 3. 3 → 1 0 0 0 0 0 0
- 5. 5 → 1 0 1 0 0 0 0

d ed e. Nessuna delle due ha avuto riferimenti:

- 3. 3 → 0 0 1 0 0 0 0
- 5. 5 → 0 0 1 0 1 0 0

Con NFU è Aging, la pagina 3 viene rimossa poiché la pagina 5 ha avuto riferimenti in (a) prima e la pagina 3 no. L'uso di un contatore a 16, 32 o 64 bit fornisce una cronologia più lunga, al costo di una maggior quantità di memoria per contenerla. Di solito 8 bit sono più che sufficienti.

8. Working Set

Demand Paging

Quando i processi sono avviati, nessuna delle loro pagine è in memoria. Appena la CPU prova a prelevare la prima istruzione, genera un page fault e fa sì che il sistema operativo prelevi la pagina con la prima istruzione. Rapidamente seguono altri page faults per le variabili globali e lo stack. Dopo un certo tempo, il processo ha la maggior parte delle pagine che gli servono ed è in condizione di essere eseguito con relativamente pochi page faults.

Working Set

L'insieme delle pagine che un processo sta attualmente usando. Se l'intero set di lavoro è in memoria, il processo sarà eseguito senza causare molti page faults fino a quando passerà a un'altra fase dell'esecuzione.

Molti sistemi di paginazione cercano quindi di tenere traccia del set di lavoro di ciascun processo e di accertarsi che sia in memoria prima di consentirne l'esecuzione. Tale approccio è chiamato *working set model* ed è stato progettato per ridurre notevolmente la frequenza dei page faults. Il caricamento delle pagine prima di lasciar eseguire i processi è detto anche prepaginazione o prepagina.

È risaputo che raramente i programmi fanno riferimenti a tutto il loro spazio degli indirizzi uniformemente; i riferimenti tendono a raggrupparsi su un numero ristretto di pagine. In ogni istante t esiste un insieme che consiste di tutte le pagine usate dai k riferimenti alla memoria più

recenti. Questo insieme $w(k, t)$ è il set di lavoro. $w(k, t)$ è una funzione monotona non decrescente di k al crescere di k perché le pagine o aumentano oppure il numero rimane costante se non servono al processo. Il limite di $w(k, t)$ quando k diventa grande è finito, poiché un programma non può fare riferimento a più pagine di quante ne contenga il suo spazio degli indirizzi e pochi programmi utilizzeranno ogni singola pagina.

Se il working set di un processo è completamente in memoria, si verificano pochi page faults. Se il working set è più grande della memoria disponibile, si verificano frequenti page faults, rallentando significativamente il processo. Questo fenomeno è noto come *thrashing*, ovvero la CPU è occupata solo a scambiare le pagine da memoria volatile a memoria non volatile.

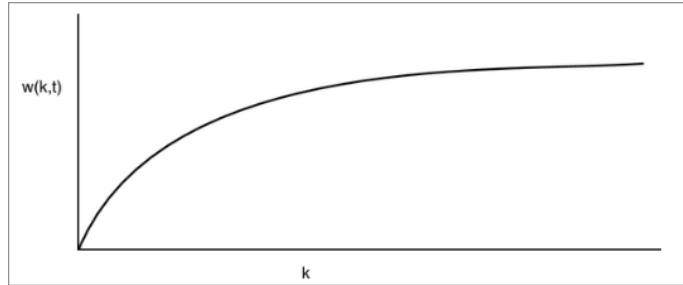


Figure 4: Tracciamento del Working Set.

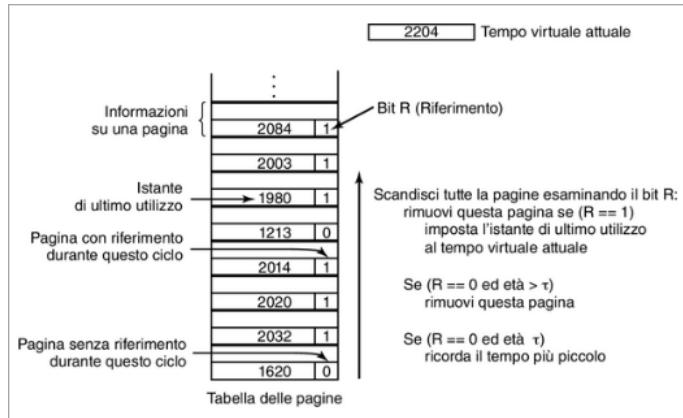


Figure 5: Algoritmo basato su working set.

Andiamo adesso ad analizzare un algoritmo di sostituzione delle pagine che si basa sul set di lavoro. L'idea base è quella di trovare una pagina che non sia nel set di lavoro e rimuoverla. Un interrupt periodico azzera il bit R a ogni ciclo di clock.

Cosa succede durante un page fault? Viene fatta una scansione delle pagine alla ricerca di una pagina da rimuovere. Viene controllato il bit R di ciascuna pagina.

- Se $R = 1$, viene aggiornato il tempo dell'ultimo utilizzo, la pagina rimane nel working set.
- Se $R = 0$ e $\text{eta} > t$, la pagina non è nel working set e viene rimossa.

- Se $R = 0$ e $\eta \leq t$, la pagina è nel working set e non viene rimossa.

Se nessuna pagina è rimovibile, viene selezionata la più vecchia con $R = 0$, altrimenti una a caso.

9. WSClock

WSClock è un miglioramento dell'algoritmo di Clock che integra le informazioni del working set. Usa una lista circolare di frame, simile all'algoritmo Clock. Ogni frame nella lista contiene: il tempo di utilizzo, il bit R e il bit M.

Ad ogni page fault è esaminata per prima la pagina indicata dalla lancetta dell'orologio. Se il bit R = 1 la pagina non è la candidata ideale alla rimozione, ovvero è stata usata nel ciclo del clock, poi il bit R viene impostato a 0. La lancetta avanza alla pagina successiva e l'algoritmo viene ripetuto per la nuova pagina.

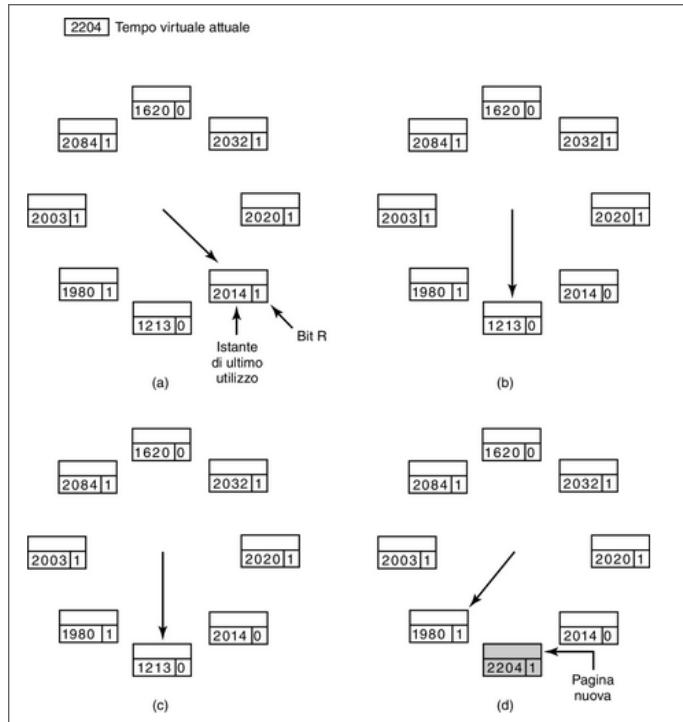


Figure 6: WSClock.

Se la pagina indicata ha $R = 0$ e se $\eta > t$:

- Se $M = 0$ (pagina pulita), non è nel working set e ne esiste una copia valida su memoria non volatile. Quindi il frame viene semplicemente riciclato e vi viene posta la nuova pagina.
- Se $M = 1$ (pagina sporca), non ne esiste una copia valida in memoria non volatile quindi non può essere sfrattata immediatamente.

Per evitare "rallentamenti", la scrittura su memoria non volatile viene schedulata e rimandata. Lungo la lista potrebbe esserci una pagina pulita e vecchia che può essere usata immediatamente. La lancetta avanza e l'algoritmo procede con la pagina successiva.

In conclusione, quali sono i migliori algoritmi? Aging e WSClock sono i migliori tra gli algoritmi analizzati, entrambi basati rispettivamente su LRU e sull'idea di Working Set, con buone prestazioni e implementazione efficiente. La nozione di migliore è il risultato del trade-off tra la complessità del metodo e i vincoli hardware che il sistema operativo deve comunque rispettare.

Sistemi Operativi

Ionut Zbirciog

16 November 2023

MEMORIA VIRTUALE

Mentre da un lato i registri base e limite possono essere utilizzati per creare l'astrazione degli spazi degli indirizzi, dall'altro sorge un nuovo problema: la gestione del bloatware, cioè del software sempre più "gonfio". Le dimensioni della memoria sono in costante aumento, ma quelle dei software aumentano ancora più velocemente.

L'idea alla base della memoria virtuale è che ogni programma ha un suo spazio degli indirizzi, suddiviso in parti chiamati pagine. Ogni pagina è un intervallo di indirizzi contigui. Queste pagine sono mappate sulla memoria fisica, ma per eseguire il programma non è indispensabile che tutte le pagine siano contemporaneamente nella memoria fisica. Quando il programma fa riferimento a una parte del suo spazio degli indirizzi che è nella memoria fisica, l'hardware esegue direttamente la mappatura necessaria. Quando il programma fa riferimento a una parte del suo spazio degli indirizzi che non è nella memoria fisica, il sistema operativo viene allertato, va a prelevare la parte mancante ed esegue nuovamente fallita.

Implementazioni diverse della memoria virtuale adottano scelte diverse rispetto a queste unità; oggi la maggior parte dei sistemi impiega una tecnica detta paging (paginazione) che prevede unità di dimensioni fisse, ad esempio di 4 KB. Una soluzione alternativa detta segmentazione usa come unità interi segmenti di dimensione variabile.

PAGINAZIONE

La maggior parte dei sistemi di memoria virtuale usa una tecnica chiamata paginazione o paging. Su qualsiasi computer i programmi fanno riferimento a un set di indirizzi di memoria, come ad esempio quando un programma esegue un'istruzione come `MOV REG,1000`.

`MOV REG,1000`

Gli indirizzi generati dal programma sono chiamati indirizzi virtuali e formano lo spazio degli indirizzi virtuali. Sui computer senza memoria virtuale, l'indirizzo virtuale è posto direttamente sul bus di memoria e provoca la lettura o la scrittura della parola della memoria fisica con lo stesso indirizzo. Quando è utilizzata la memoria virtuale, gli indirizzi virtuali non vanno direttamente al bus di memoria, ma a una MMU (Memory Management Unit, unità di gestione della memoria) che mappa gli indirizzi virtuali sugli indirizzi della memoria fisica (circuito presente nella CPU).

Lo spazio degli indirizzi virtuali è suddiviso in unità di dimensione fissa, chiamate pagine. Le unità corrispondenti nella memoria fisica sono chiamate frame o page frame. Le pagine e i frame

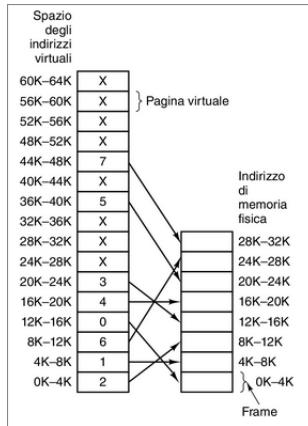


Figure 1: Una tabella delle pagine con 16 pagine e 8 frame

sono della stessa dimensione, ad esempio da 4 KB. Nei sistemi reali le pagine vanno da 512 byte fino a 64 KB.

- 64 Kb di pagine virtuali possono essere mappate in 8 frame, usando 16 bit per indirizzo.
- I trasferimenti fra RAM e memoria non volatile sono sempre di pagine intere. Molti processori supportano più dimensioni di pagina, che possono essere mescolati e abbinati a discrezione del sistema operativo.
- Ad esempio, l'architettura x86-64 supporta pagine da 4 KB, 2 MB e 1-GB, e potremmo utilizzare le pagine da 4 KB per le applicazioni utente e una singola pagina da 1 GB per il kernel.

Esempi

1. L'istruzione `MOV REG, 8192` è trasformata effettivamente in `MOV REG, 24576` poiché l'indirizzo virtuale 8192 (nella pagina virtuale 6) è mappato sul 24576 (nel frame fisico 24K–28K).
2. L'indirizzo virtuale 20500 dista 20 byte dall'inizio della pagina virtuale 3 (indirizzi virtuali da 20480 a 24575) e la sua mappatura sull'indirizzo fisico è $12288 + 20 = 12308$.

Questa capacità di mappare le 16 pagine virtuali su uno qualsiasi degli 8 frame, impostando in modo appropriato la mappa della MMU, da sola non basta a risolvere il problema dello spazio degli indirizzi virtuali più grande della memoria fisica. Poiché abbiamo solo 8 frame fisici, solo otto delle pagine virtuali sono mappate sulla memoria fisica. Le altre, mostrate nella figura con una X, non sono mappate. Nell'hardware, un bit presente/assente tiene traccia di quali pagine sono presenti fisicamente in memoria.

Che cosa accade se, per esempio, il programma fa riferimento a indirizzi non mappati usando l'istruzione `MOV REG, 32780` che è il byte 12 all'interno della pagina virtuale 8 (che parte da 32768)? La MMU rileva che la pagina non è mappata (è contrassegnata da una X nella figura) e causa una trap della CPU verso il sistema operativo. Questa trap è chiamata page fault (errore di pagina).

Il sistema operativo preleva un frame poco utilizzato e ne scrive il contenuto su disco (se non è già presente). Prende poi la pagina alla quale è stato appena fatto riferimento e la pone nel frame appena liberato, cambia la mappa e riavvia l'istruzione che era in trap.

MMU

Guardiamo adesso all'interno della MMU per vedere come funziona e perché abbiamo scelto di usare una dimensione di pagina che sia una potenza di 2.

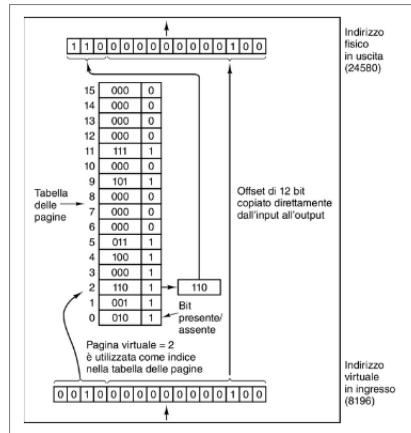


Figure 2: MMU

Nella figura viene mostrato un esempio di un indirizzo virtuale, 8196 (001000000000100 in binario), mappato usando la mappa della MMU mostrata in figura. L'indirizzo virtuale di 16 bit in ingresso è suddiviso in un numero di pagina di 4 bit e un offset di 12 bit. Con 4 bit per il numero di pagina (bit usati per la conversione da indirizzo virtuale a indirizzo fisico), possiamo avere 16 pagine (2^4 pagine) e con 12 bit di offset (bit usati per scorrere all'interno della pagina) possiamo indirizzare 4096 byte (4Kb) per pagina, ovvero $4\text{Kb} * 16$ pagine = 64Kb per ogni processo.

Il numero di pagina è usato come indice nella tabella delle pagine che porta al numero di frame corrispondente alla pagina virtuale. Se il bit presente/assente è 0, avviene una trap al sistema operativo. Se il bit è 1, il numero di frame trovato nella tabella delle pagine viene copiato nei tre bit più significativi del registro di output, insieme all'offset di 12 bit che è copiato senza modifiche dall'indirizzo virtuale in arrivo. Insieme formano un indirizzo fisico di 15 bit. Il registro di output è poi posto sul bus di memoria come indirizzo fisico di memoria.

STRUTTURA DI UNA VOCE NELLA TABELLA DELLE PAGINE

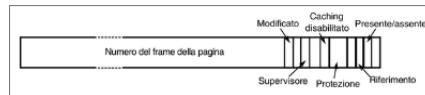


Figure 3: Struttura di una voce nella tabella delle pagine

Nella figura è mostrato l'esempio di una voce della tabella delle pagine. La dimensione cambia a seconda del computer, ma quella usuale in un generico computer odierno è di 64 bit. Il campo più importante è il numero del frame (frame number): l'obiettivo della mappatura delle pagine è ottenere questo valore. Se la dimensione della pagina è 4 KB, per il numero del frame sono necessari solo i 52 bit più significativi, e rimangono 12 bit per codificare altre informazioni sulla pagina. Ad esempio, il bit Presente/assente indica se la voce è valida e possa essere usata. Se questo bit è 0, la pagina virtuale cui appartiene la voce non è effettivamente in memoria. L'accesso alla voce della tabella delle pagine con questo bit impostato a 0 causa un page fault.

I bit Protezione specificano quali tipi di accesso sono consentiti. Nella forma elementare questo campo contiene 1 bit, con 0 che significa lettura/scrittura e 1 per la sola lettura. Un'impostazione più sofisticata ha 3 bit, ogni bit per consentire lettura, scrittura ed esecuzione della pagina. Il bit Supervisor è in qualche modo correlato, e indica se la pagina sia accessibile soltanto al codice con privilegi, ovvero al sistema operativo (o supervisore) oppure anche ai programmi utente. Qualsiasi tentativo di accesso a una pagina supervisore da parte di un programma utente causa un page fault.

I bit Modificato e Riferimento tengono traccia dell'uso della pagina. Quando viene scritta una pagina, l'hardware impone automaticamente il bit Modificato. Questo bit è valorizzato quando il sistema operativo decide di riutilizzare un frame. Se la sua pagina è stata modificata (cioè è "sporca") deve essere riscritta sulla memoria non volatile. Se non è stata modificata (cioè è "pulita") può essere abbandonata, poiché la copia su disco o SSD è ancora valida. Talvolta il bit è chiamato dirty bit poiché riflette lo stato della pagina.

Il bit Riferimento è impostato ogni qualvolta si faccia riferimento alla pagina, sia in lettura sia in scrittura. Serve per aiutare il sistema operativo a scegliere una pagina da "sfrattare" quando si verifica un page fault; le pagine inutilizzate sono più "sfrattabili" di quelle usate, e questo bit gioca un ruolo importante in molti degli algoritmi di sostituzione delle pagine che studieremo in seguito.

Per un processo, l'indirizzo in memoria della "sua" tabella delle pagine è scritto nel registro Page Table Base Register.

COME VELOCIZZARE LA PAGINAZZIONE

Problema

1. La mappatura dall'indirizzo virtuale all'indirizzo fisico deve essere veloce;
2. Anche se lo spazio virtuale degli indirizzi è enorme, la tabella delle pagine non deve essere troppo grande.

Soluzioni

1. **Tabella delle pagine in Registri Hardware:** All'avvio di un processo, il sistema operativo carica i registri con la tabella delle pagine del processo, presa da una copia che tiene in memoria. Durante l'esecuzione del processo non sono necessari altri riferimenti alla memoria per la tabella delle pagine. Il vantaggio di questo metodo è che è semplice e non richiede riferimenti alla memoria durante la mappatura. Lo svantaggio è che è insopportabilmente dispendioso se la tabella delle pagine è estesa; nella maggior parte dei casi non è praticabile. Un altro è che dover caricare l'intera tabella delle pagine a ogni cambio di contesto compromette le prestazioni.

2. **Tabella delle pagine in Memoria RAM:** Questo design consente di cambiare la mappatura virtuale-fisica a ogni cambio di contesto ricaricando un solo registro. Lo svantaggio è che naturalmente richiede uno o più riferimenti di memoria per la lettura della tabella delle pagine durante l'esecuzione di ogni istruzione, rendendola molto lenta.

TLB (Translation Lookaside Buffer)

Ogni istruzione richiede l'accesso alla memoria per prelevare l'istruzione stessa e un ulteriore accesso per la tabella delle pagine. Questo raddoppio degli accessi alla memoria riduce le prestazioni di metà.

I progettisti di computer hanno escogitato una soluzione basata sull'osservazione che la maggior parte dei programmi tende a fare un gran numero di riferimenti a un piccolo numero di pagine e non il contrario. Dunque solo una piccola parte delle voci della tabella delle pagine viene letta frequentemente; il resto è poco utilizzato.

La soluzione è dotare i computer di un piccolo dispositivo hardware per mappare gli indirizzi virtuali sugli indirizzi fisici senza passare dalla tabella delle pagine. Il dispositivo è chiamato TLB (Translation Lookaside Buffer) o qualche volta anche memoria associativa. Si trova abitualmente all'interno della MMU e consiste di un numero ridotto di voci, otto in questo caso, ma raramente più di 256. Ciascuna voce contiene informazioni riguardo una pagina, tra cui il numero di pagina virtuale, un bit impostato quando la pagina viene modificata, il codice di protezione (i permessi di lettura, scrittura ed esecuzione) e il frame fisico in cui si trova la pagina. Questi campi hanno una corrispondenza uno-a-uno con i campi nella tabella delle pagine, eccetto che per il numero di pagina virtuale, che non è necessario nella tabella delle pagine. Un altro bit indica se la voce è valida (cioè in uso) o no.

| Valido | Pagina virtuale | Modificato | Protezione | Frame |
|--------|-----------------|------------|------------|-------|
| 1 | 140 | 1 | RW | 31 |
| 1 | 20 | 0 | R X | 38 |
| 1 | 130 | 1 | RW | 29 |
| 1 | 129 | 1 | RW | 62 |
| 1 | 19 | 0 | R X | 50 |
| 1 | 21 | 0 | R X | 45 |
| 1 | 860 | 1 | RW | 14 |
| 1 | 861 | 1 | RW | 75 |

Figure 4: TLB con 8 voci

Funzionamento della TLB

Quando un indirizzo virtuale viene presentato alla MMU per la traduzione, l'hardware prima guarda se il suo numero di pagina virtuale è presente nel TLB confrontandolo simultaneamente (cioè in parallelo) con tutte le voci. Questa operazione richiede un hardware specializzato, presente in tutte le MMU dotate di TLB. Se trova un riscontro valido e l'accesso non viola i bit di protezione, il frame è prelevato direttamente dal TLB, senza andare alla tabella delle pagine in memoria. Se il numero di pagina virtuale è presente nel TLB, ma l'istruzione prova a scrivere su una pagina di sola lettura, si genera un errore di protezione (Segmentation Fault).

È interessante ciò che accade quando il numero di pagina virtuale non è nel TLB. La MMU rileva il TLB miss (assenza dal TLB) ed esegue una normale ricerca sulla tabella delle pagine. Quindi sfratta una delle voci dal TLB e la rimpiazza con la voce della tabella delle pagine appena trovata, così se quella pagina viene riutilizzata a breve, la seconda volta si avrà una TLB hit (presenza nel TLB) invece di un TLB miss. Quando una voce è eliminata dal TLB, il bit Modificato viene copiato di nuovo nella voce della tabella delle pagine nella memoria. Gli altri valori sono già lì, eccetto il bit Riferimento. Quando il TLB viene caricato dalla tabella delle pagine, tutti i campi vengono presi dalla memoria.

La modifica dei bit di accesso ad una pagina richiedono l'aggiornamento del TLB. Per garantire la coerenza, la voce corrispondente nel TLB viene eliminata o aggiornata.

Alcune architetture RISC come SPARC, MIPS e HP PA gestiscono le voci del TLB a livello software. Quando si verifica un TLB miss, invece di far andare la MMU alla tabella delle pagine per cercare e prelevare il necessario riferimento alla pagina, viene semplicemente generato un errore di TLB e il problema passa al sistema operativo. Il sistema deve trovare la pagina, rimuovere una voce dal TLB, inserirne una nuova e riavviare l'istruzione in errore. Naturalmente tutto questo deve avvenire con una manciata di istruzioni, poiché i TLB miss sono molto più frequenti dei page fault, ed è importante capire il perché.

Quanti tipi ci sono di TLB MISS e quali sono?

1. **Soft Miss:** la pagina di riferimento non è nel TLB ma è nella memoria. In questo caso basta aggiornare il TLB, non serve alcun I/O da disco o SSD. Solitamente per trattare un soft miss ci vogliono 10-20 istruzioni macchina che possono essere completate in pochi nanosecondi.
2. **Hard Miss:** la pagina richiesta non è in memoria (e ovviamente nemmeno nel TLB). Per prelevare la pagina serve un accesso al disco o all'SSD, nell'ordine dei millisecondi a seconda del tipo di memoria non volatile. Un hard miss è milioni di volte più lento di un soft miss.

Cercare la mappatura nella gerarchia delle tabelle delle pagine è un'operazione che prende il nome di *page table walk* (passeggiata per la tabella delle pagine).

Un miss non è solamente soft o hard; alcuni miss sono leggermente più soft (o leggermente più hard) rispetto ad altri. Si supponga, per esempio, che la page walk non trovi la pagina all'interno della tabella delle pagine del processo e il programma incorra in un errore di pagina. Le possibilità sono tre.

1. La pagina potrebbe essere in memoria, ma non nella tabella delle pagine di questo processo, magari perché è stata caricata dalla memoria non volatile da parte di un altro processo. In questo caso non è necessario accedere nuovamente alla memoria non volatile, ma è sufficiente mappare la pagina in maniera appropriata nella tabella delle pagine. Questo è un miss abbastanza soft, chiamato anche *minor page fault*.
2. Un *major page fault* si può verificare se la pagina dev'essere caricata dalla memoria non volatile.
3. Il programma abbia semplicemente avuto accesso a un indirizzo non valido e non sia necessario aggiungere alcuna mappatura al TLB. In questo caso, il sistema operativo solitamente termina il programma con un segmentation fault.

DIMENSIONE DELLE TABELLE DELLE PAGINE

Problema: con 32 bit si hanno 2^{20} pagine per ogni processo, con 64 bit si hanno 2^{32} pagine per ogni processo, che porta ad uno spreco di memoria perché si ha un enorme spazio di indirizzi e un enorme tabella delle pagine che bisogna tenere in memoria.

Soluzione: Tabelle Multi livello. Un semplice esempio è illustrato nella figura sotto. Abbiamo un indirizzo virtuale a 32 bit partizionato in un campo PT1 a 10 bit, un campo PT2 a 10 bit e un campo Offset a 12 bit. Poiché gli offset sono 12 bit, le pagine sono da 4 KB e ce ne sono in totale 2^{20} . Il segreto del metodo della tabella delle pagine multilivello sta nell'evitare di mantenere tutte le tabelle delle pagine in memoria per tutto il tempo. In particolare quelle non necessarie dovrebbero essere messe da parte (viene creata una nuova tabella quando è necessario).

CR3 register: Registro speciale per puntare al vertice della gerarchia delle tabelle di pagina.

Con 64 bit si hanno tabelle di pagine a 4 livelli. I processori oggi utilizzano tutte le 512 voci di ciascuna tabella, arrivando a una quantità di memoria indirizzabile pari a $2^9 \times 2^9 \times 2^9 \times 2^9 \times 2^{12} = 2^{48}$ byte. Si poteva aggiungere anche un altro livello, ma probabilmente i produttori hanno pensato che, almeno per il momento, 256 TB di memoria indirizzabile sarebbero stati sufficienti.

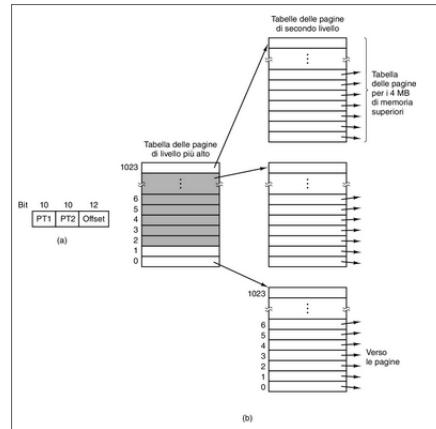


Figure 5: Tabella multi livello

Sistemi Operativi

Ionut Zbirciog

21 November 2023

1 Problemi di Progettazione dei Sistemi di Paginazione

Allocazione Globale vs Locale

Allocazione Locale

Ogni processo riceve una porzione fissa della memoria. Quando avviene un page fault, l'algoritmo di sostituzione delle pagine andrà a sfrattare una delle pagine allocate al processo. Semplice da implementare, ma può portare a inefficienze se il set di lavoro aumenta o diminuisce, portando al thrashing.

Allocazione Globale

La memoria viene distribuita in modo dinamico tra i processi. Quando avviene un page fault, l'algoritmo di sostituzione delle pagine può scegliere di sfrattare anche una pagina allocata a un processo diverso. È più efficace per adattarsi alle esigenze variabili dei processi, ovvero quando la dimensione del set di lavoro varia nel tempo, ma richiede una gestione più complessa. Con l'allocazione globale, il sistema operativo deve dinamicamente assegnare e riassegnare frame ai processi. È possibile usare i bit di aging per monitorare la frequenza di accesso alle pagine, anche se non sono sufficienti contro il thrashing, poiché offrono una stima approssimativa che potrebbe non riflettere cambiamenti rapidi nel set di lavoro.

Un altro approccio è di avere un algoritmo per allocare frame ai processi assegnando un numero uguale di frame a ciascun processo (allocazione equa). Anche se sembra equo, in realtà non lo è perché significherebbe assegnare lo stesso numero di pagine ad un processo da 10KB e ad uno di 300KB. Sarebbe più opportuno allocare le pagine in proporzione della dimensione del processo (allocazione proporzionale). Questa soluzione rispecchia meglio la necessità di memoria, evitando allocazioni inadeguate.

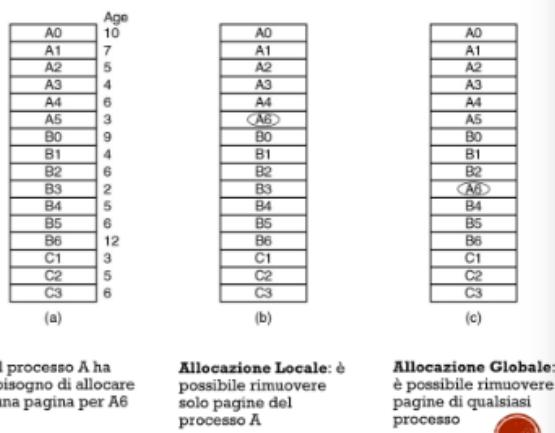


Figure 1: Allocazione statica vs Allocazione dinamica

Usando un algoritmo di allocazione dinamico è possibile avviare ogni processo con un certo numero di pagine proporzionale alla sua dimensione che dovranno essere gestite in modo dinamico in base alle esigenze del processo. Un modo per gestire l'allocazione è usare l'algoritmo PFF (Page Fault Frequency), monitora la frequenza dei page fault per regolare l'allocazione di memoria di un processo, aumentando i frame se i page fault sono troppo frequenti, diminuisce se sono rari.

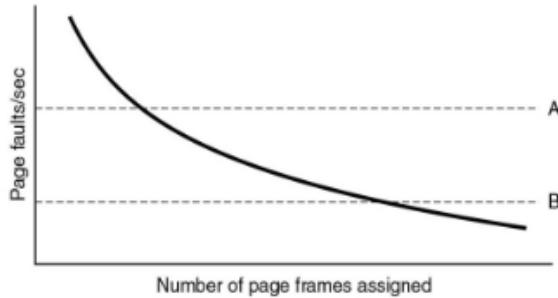


Figure 2: Numero di pagine assegnate

- A: Alta frequenza di page fault indica necessità di più frame.
- B: Bassa frequenza di page fault suggerisce che il processo ha più memoria del necessario.

Controllo del Carico

Anche con il miglior algoritmo, il thrashing può sempre verificarsi se i set di lavoro di tutti i processi eccedono la memoria disponibile.

- **Out Of Memory Killer (OOM):** processo di sistema che seleziona e termina i processi in base a un punteggio di "cattiveria" per liberare memoria. Sono selezionati di solito i processi con elevato utilizzo della memoria e minor importanza.
- **Swapping:** sposta i processi su memoria non volatile, liberando le loro pagine per altri processi, riducendo la richiesta di memoria senza interrompere i processi. Solo una parte dei processi in memoria non volatile è schedulata attivamente, aiutando a gestire meglio il carico della memoria; è utile anche per ridurre l'occupazione della memoria da processi eseguiti in background.
- Oltre a uccidere o spostare processi, si possono usare compattamento, compressione o deduplicazione.

Policy di Pulizia

Per garantire una certa abbondanza di pagine libere, i sistemi dispongono di un processo in background detto *paging daemon*, che è a riposo per la maggior parte del tempo ma viene risvegliato periodicamente per ispezionare lo stato della memoria. Quando i frame liberi scarseggiano, inizia a selezionare pagine da rimpiazzare utilizzando un algoritmo di sostituzione delle pagine. Un modo per implementare questa policy di pulizia è attraverso un orologio a due lancette. La lancetta anteriore (gestita dal daemon paging) avanza scrivendo le pagine sporche in memoria non volatile. La lancetta posteriore si occupa della sostituzione delle pagine.

Dimensione delle Pagine

Vantaggi Pagine Piccole

Riduzione della frammentazione interna e dell'utilizzo di memoria, poiché un programma potrebbe richiedere meno memoria con pagine piccole.

Svantaggi Pagine Piccole

Richiedono tabelle delle pagine più grandi e possono aumentare il tempo e lo spazio necessari per il trasferimento di dati e la gestione della memoria.

Alcuni sistemi operativi utilizzano pagine di diverse dimensioni per parti diverse del sistema, ad esempio, pagine più grandi per il kernel. Alcuni sistemi operativi effettuano salti mortali spostando la memoria di un processo per creare intervalli contigui adatti a una pagina grande (THP, Transparent Huge Pages).

Dimensione Ottimale e Calcolo della Dimensione Ottimale

La dimensione ottimale è determinata bilanciando la frammentazione interna e l'overhead della tabella delle pagine. Bisogna tenere conto della dimensione media del processo (s byte), della dimensione della pagina (p byte) e della dimensione di ogni voce nella tabella delle pagine (e byte).

L'overhead è dato da $\frac{s}{p} + \frac{p}{2}$, dove il primo termine (tabella delle pagine) aumenta con pagine più piccole, e il secondo termine (frammentazione interna) aumenta con pagine più grandi. L'ottimo si trova bilanciando questi due fattori. Derivando la funzione di overhead rispetto a p e ponendola uguale a zero si ottiene: $-\frac{s}{p^2} + \frac{1}{2} = 0$. Da cui si ottiene la dimensione ottimale $p = \sqrt{2se}$. Per esempio, con $s = 1$ MB e $e = 8$ byte, la dimensione ottimale è 4 KB, la dimensione comune attuale.

Istruzioni Separate e Spazi di Dati

È comune che molti utenti eseguano lo stesso programma o utilizzino le stesse librerie. Condividere pagine di memoria tra questi processi è più efficiente che mantenere copie separate. Per facilitare la condivisione, è meglio separare lo spazio degli indirizzi di un processo in:

- **I-Space:** spazio degli indirizzi delle istruzioni.
- **D-Space:** spazio degli indirizzi dei dati.

Ogni spazio degli indirizzi va da 0 a un certo massimo, tipicamente $2^{16} - 1$ o $2^{32} - 1$. Il linker deve sapere quando sono utilizzati I-Space e D-Space separatamente, poiché in questo caso i dati sono riposizionati all'indirizzo virtuale 0 anziché partire dopo il programma.

Pagine e Librerie Condivise

Condivisione delle Pagine

È possibile condividere solo le pagine di sola lettura, come il testo dei programmi e non le pagine dei dati. Processi diversi possono utilizzare la stessa tabella delle pagine per l'I-Space, ma tabelle diverse per il D-Space. Ciascun processo ha puntatori sia all'I-Space che al D-Space. Lo scheduler utilizza questi puntatori per impostare l'MMU. Questo meccanismo di condivisione delle pagine ha un grande problema quando un processo viene rimosso, può causare numerosi page fault in un altro processo che condivide le stesse pagine. In UNIX, dopo l'esecuzione di un fork, genitore e figlio condividono sia il testo che i dati, inizialmente come sola lettura. Se un processo modifica i dati, si genera una trap, e viene creata una copia della pagina modificata (*Copy on Write*). Questo metodo evita la copia di pagine che non vengono modificate.

Condivisione delle Librerie

I sistemi operativi condividono automaticamente tutte le pagine di testo di un programma avviato più volte. Se un processo modifica una pagina di dati condivisa, occorre applicare "copy on write". Nei sistemi operativi, esistono molte grandi librerie usate da molti processi, ad esempio, librerie di I/O e grafiche. Perciò collegare tutte queste librerie a ogni programma eseguibile lo renderebbe ancora più ingombrante di quanto già lo sia. Una tecnica è quindi usare librerie condivise, chiamate DLL (Dynamic Link Library) in Windows o file .so (Shared Objects) in UNIX.

Le librerie condivise possono essere posizionate a indirizzi diversi nei vari processi. Questo impedisce l'uso di indirizzi assoluti nelle istruzioni. Pertanto, le librerie condivise vengono compilate con indirizzi relativi anziché assoluti.

File Mappati in Memoria

I file mappati in memoria si riferiscono a una tecnica utilizzata nei sistemi operativi per consentire ai processi di accedere direttamente ai file su disco come se fossero memorizzati nella memoria principale del computer. Questa tecnica è comunemente utilizzata per ottimizzare l'accesso ai dati e migliorare le prestazioni del sistema. Quando un file viene mappato in memoria, una porzione del file viene associata a una regione di memoria virtuale. In pratica, il sistema operativo crea un collegamento tra il file su disco e una porzione della memoria virtuale del processo. Ciò consente al processo di accedere direttamente ai dati del file come se fossero presenti in memoria, senza dover leggere o scrivere direttamente dal disco ogni volta che è necessario accedere ai dati. I file mappati forniscono un modello alternativo per l'I/O. Anziché eseguire letture e scritture, si può accedere al file come a un grande array di caratteri in memoria.

L'utilizzo dei file mappati in memoria può portare a una maggiore efficienza nell'accesso ai dati, poiché il sistema operativo può gestire in modo più intelligente il caricamento e il salvataggio dei dati tra la memoria e il disco. Inoltre, i file mappati in memoria consentono di condividere dati tra processi, poiché più processi possono mappare la stessa regione di memoria associata a un file.

2 Problemi di Implementazione

Il Sistema Operativo e la Paginazione

I momenti in cui il sistema operativo deve svolgere attività correlate alla paginazione sono quattro:

Creazione del Processo

Il sistema operativo deve determinare le dimensioni iniziali del programma e dei dati, creare e inizializzare la tabella delle pagine, allocare spazio nella memoria non volatile per lo scambio, inizializzare l'area di scambio e registrare informazioni nella tabella dei processi.

Esecuzione del Processo

Il sistema operativo deve azzerare la MMU, se necessario svuotare la TLB, rendere attiva la tabella delle pagine del processo e, optionalmente, caricare alcune pagine in memoria per ridurre i page fault iniziali (pre-paginazione).

Gestione dei Page Fault

Il sistema operativo deve determinare l'indirizzo virtuale che ha causato il page fault, trovare la pagina necessaria nella memoria non volatile, scegliere un frame disponibile, eventualmente sfrattando pagine vecchie, caricare la pagina nel frame e ripristinare il contatore di programma.

Chiusura del Processo

Il sistema operativo deve rilasciare la tabella delle pagine, le pagine in memoria e lo spazio su memoria non volatile, gestire le pagine condivise con altri processi, rilasciandole solo dopo l'ultimo utilizzo.

Gestione dei Page Fault

1. L'hardware esegue la trap nel kernel, salvando il contatore del programma nello stack. Nella maggior parte delle macchine, alcune informazioni sullo stato dell'istruzione corrente vengono salvate in registri speciali della CPU.
2. Viene avviata una routine di servizio interrupt in codice assembly che salva i registri e altre informazioni volatili per evitare che il sistema operativo li elimini, poi chiama il gestore dei page fault.
3. Il sistema operativo determina quale pagina virtuale manca, se non disponibile nei registri hardware, recuperando la pagina analizzando l'istruzione dal program counter.
4. Il sistema controlla che l'indirizzo sia valido e che la protezione sia coerente con l'accesso. Se non lo è, viene mandato un segnale al processo o quest'ultimo viene terminato. Se l'indirizzo è valido e non è avvenuto alcun errore di protezione, il sistema verifica se c'è un frame libero. Se non vi sono frame liberi, viene eseguito un algoritmo di sostituzione delle pagine per liberarne uno.
5. Se la pagina è sporca, viene schedulata per la scrittura in memoria non volatile e il processo è sospeso consentendo l'esecuzione di un altro processo nel mentre avviene il swapping.
6. Una volta liberato, il frame viene usato per caricare la pagina necessaria da disco o SSD. Durante il caricamento della pagina, il processo in page fault è ancora sospeso e viene eseguito, se disponibile, un altro processo utente.
7. Quando l'interrupt del disco o dell'SSD indica che è arrivata la pagina, le tabelle delle pagine vengono aggiornate in modo da riflettere la sua posizione e il frame è contrassegnato in stato normale.
8. L'istruzione in errore è riportata allo stato che aveva all'inizio e il contatore di programma è ripristinato in modo da puntare a quell'istruzione.
9. Il processo in errore è schedulato per l'esecuzione e il sistema operativo torna alla routine (in linguaggio assembly) che lo aveva invocato.
10. La routine in questione ricarica i registri e le altre informazioni di stato e ritorna allo spazio utente per riprendere l'esecuzione da dove si era interrotta.

Bloccare le Pagine in Memoria

Un processo invia una richiesta di lettura da un file o dispositivo in un buffer nel suo spazio degli indirizzi. Mentre attende il completamento dell'I/O, può essere sospeso per permettere l'esecuzione di un altro processo.

Se il secondo processo genera un page fault, esiste il rischio che la pagina contenente il buffer di I/O venga selezionata per essere rimossa. Una soluzione a questo problema è di bloccare o "pinnare" in memoria le pagine utilizzate per l'I/O, assicurando che le operazioni possano procedere senza interruzioni. Oppure gestire l'I/O nei buffer del kernel e poi copiare i dati nelle pagine utente.

Memoria Secondaria

Il sistema operativo prevede una partizione speciale o dispositivo separato per lo scambio, come nei sistemi UNIX. Un'area del disco/SSD strutturata in maniera differente dai file system. Ogni processo ha un'area di scambio in memoria non volatile. Per affrontare la crescita dinamica di un processo, vengono riservate aree separate per testo, dati e stack. In alternativa, l'allocazione su disco/SSD avviene al momento dello scambio di ogni pagina.

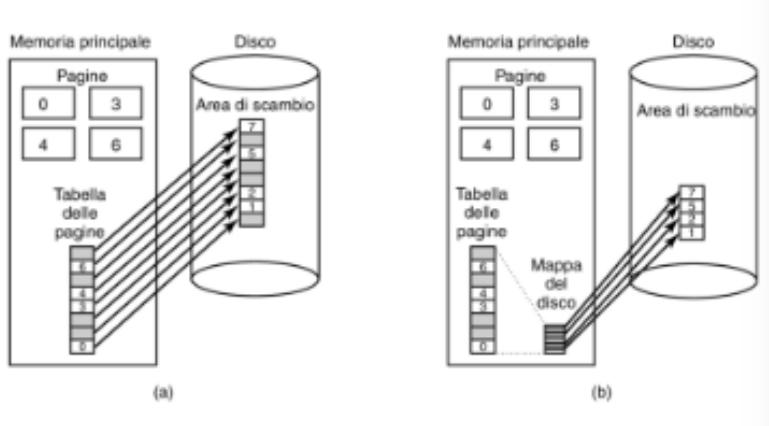


Figure 3: Struttura dell'area di scambio su disco/SSD.

3 Segmentazione

In un sistema di memoria monodimensionale, gli indirizzi virtuali vanno da 0 ad un certo massimo e sono disposti in modo lineare e contiguo. Può risultare problematica in alcuni scenari, come nella compilazione, dove ci sono diverse tabelle che crescono dinamicamente e in modo prevedibile. La crescita di una tabella può causare sovrapposizioni con altre, creando difficoltà nella gestione della memoria.

La segmentazione introduce l'idea di spazi di indirizzi virtuali multipli e indipendenti, chiamati segmenti. Ciascun segmento ha una sequenza lineare di indirizzi, iniziando da 0 fino a una massima valore. I segmenti possono avere lunghezze diverse e la loro dimensione può cambiare durante l'esecuzione. Questa struttura consente ai segmenti di crescere o ridursi senza interferire l'uno con l'altro.

Vantaggi della Segmentazione

- **Flessibilità:** i segmenti possono crescere o ridursi in modo indipendente l'uno dall'altro.
- **Semplificazione del Linking:** se ogni processo occupa un segmento separato, il linking di procedure diventa molto più semplice.
- **Condivisione e Protezione:** la segmentazione facilita la condivisione di risorse, come librerie condivise, tra processi diversi. Offre anche la possibilità di applicare vari livelli di protezione ai segmenti.

Segmentazione vs Paginazione

La segmentazione suddivide la memoria in segmenti con indirizzi lineari. La segmentazione offre maggiore flessibilità e gestione delle strutture dati rispetto alla paginazione, ma può essere più complessa da implementare.

| Considerazione | Paginazione | Segmentazione |
|---|---|---|
| Il programmatore deve sapere che questa tecnica è in uso? | NO | SI |
| Quanti spazi di indirizzi lineari ci sono? | 1 | Molti |
| Lo spazio degli indirizzi totale può superare la dimensione della memoria fisica? | SI | SI |
| Le procedure e i dati possono essere distinti e protetti separatamente? | NO | SI |
| Le tabelle la cui dimensione varia possono essere disposte facilmente? | NO | SI |
| La condivisione delle procedure fra utenti è facilitata? | NO | SI |
| Perché fu inventata questa tecnica? | Per avere uno spazio degli indirizzi lineare grande senza dover acquistare ulteriore memoria fisica | Per consentire a programmi e dati di essere spezzati in spazi degli indirizzi logicamente indipendenti e per facilitare la condivisione e la protezione |

Figure 4: Confronto tra segmentazione e paginazione.

Implementazione della Segmentazione Pura

La maggior differenza tra segmentazione e paginazione è che i segmenti hanno dimensioni variabili e le pagine dimensioni fisse.

1. Memoria fisica con 5 segmenti.
2. Il segmento 1 è rimosso, e il segmento 7, che è più piccolo, viene messo al suo posto (ora tra 7 e 2 c'è dello spazio inutilizzato).
3. Il segmento 4 è sostituito dal segmento 5.
4. Il segmento 3 è rimpiazzato dal segmento 6. Ciò genera frammentazione esterna, dopo un po', la memoria sarà suddivisa in parti, alcune contenenti segmenti e altre spazi vuoti.
5. Può essere risolto tramite la compattazione.

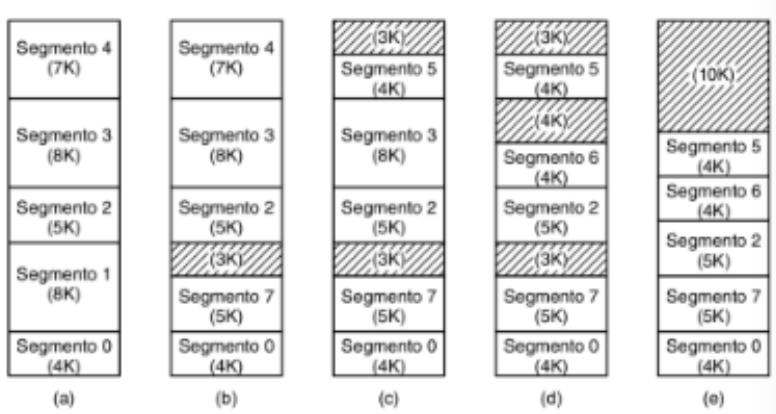


Figure 5: Esempio di Segmentazione.

Segmentazione Con Paginazione In MULTICS

In MULTICS, i segmenti venivano trattati come spazi di memoria virtuale indipendenti e paginati per gestire meglio lo spazio di memoria. C'era una tabella dei segmenti, con descrittori per ogni segmento, indicando se sono in memoria e collegamenti alle tabelle delle pagine. I descrittori avevano un puntatore al segmento, dimensione del segmento, bit di protezione e altre informazioni. Gli indirizzi erano costituiti da due parti, segmento e indirizzo del segmento, con suddivisione in numero di pagina e parola della pagina.

Conversione di un indirizzo in MULTICS

1. Il numero del segmento era usato per trovare il descrittore del segmento.
2. Si verificava se la tabella delle pagine del segmento fosse in memoria. Se lo era, veniva localizzata. Se non lo era, avveniva un errore di segmentazione (segment fault). Se c'era una violazione della protezione, si verificava un errore (trap).
3. Veniva esaminata la voce della pagina virtuale richiesta nella tabella delle pagine. Se la pagina non era in memoria veniva generato un page fault; se lo era, dalla voce della tabella delle pagine veniva estratto l'indirizzo dell'inizio della pagina nella memoria principale.
4. Si otteneva l'indirizzo nella memoria principale in cui era localizzata la parola aggiungendo l'offset all'origine della pagina.
5. Finalmente avveniva la lettura o il salvataggio.

Questo processo è stato ottimizzato con l'introduzione del TLB.

Sistemi Operativi

Ionut Zbirciog

28 November 2023

File System

I file system sono un modo per organizzare e memorizzare (in modo persistente) le informazioni. Offrono inoltre un'astrazione sui dispositivi di memorizzazione (Disco rigido, SSD, rete, RAM). Le informazioni sono organizzate in file e directory. Esempi di file system: FAT12/FAT16 per MS-DOS, NTFS per Windows, Ext4 per Linux, APFS per macOS/iOS.

File

I file sono un meccanismo di astrazione: forniscono un metodo per salvare informazioni sul disco e leggerle in seguito. Ciò deve avvenire in modo da nascondere all'utente i dettagli di come e dove le informazioni siano memorizzate e di come funziona effettivamente il disco. I file vengono identificati tramite nomi, che possono variare in base al sistema operativo. Alcuni sistemi operativi limitano la lunghezza dei file (MS-DOS) mentre altri supportano nomi più lunghi. Inoltre, alcuni sistemi come UNIX distinguono tra maiuscole e minuscole (maria != MarIa), mentre sistemi come MS-DOS no (maria == MarIa).

Ciascun file, generalmente, è identificato oltre da un nome anche da un'estensione, indicando generalmente una caratteristica specifica di un file (.jpg per immagini, .c per codice sorgente in linguaggio C, etc.). In alcuni sistemi come UNIX, le estensioni sono puramente convenzionali e non richieste dal sistema operativo. Un compilatore C, invece, potrebbe effettivamente richiedere l'estensione .c per i file da compilare, rifiutandosi di compilare se non la presentano, ma al sistema operativo poco importa. Altri sistemi come Windows, le estensioni hanno un significato specifico e sono associate a programmi specifici. Gli utenti (o i processi) possono registrare le estensioni nel sistema operativo e specificare per ognuna quale sia il programma che "possiede" quell'estensione. Quando un utente fa doppio clic sul nome di un file, il programma assegnato alla sua estensione viene lanciato con il file come parametro. Per esempio, con un doppio clic su file.docx, si avvia Microsoft Word con file.docx come file iniziale su cui lavorare. Photoshop, invece, non aprirà file con estensione .docx.

I file possono essere strutturati in tanti modi diversi. Tre delle possibilità più comuni sono descritte nella figura sotto.

Sequenza Non Strutturata di Byte

I file sono visti dal sistema operativo come una serie non strutturata di byte. Il significato dei dati è determinato dai programmi a livello utente, non dal sistema operativo. Questo approccio è adottato da sistemi come UNIX, Linux, macOS e Windows, offrendo massima flessibilità.

| Estensione | Significato |
|------------|---|
| .bak | File di backup |
| .c | Programma sorgente in linguaggio C |
| .gif | Immagine in Compuserve Graphical Interchange Format |
| .html | Documento HTML (world wide web hypertext markup language) |
| .jpg | Immagine codificata con lo standard JPEG |
| .mp3 | Musica codificata in formato audio MPEG layer 3 |
| .mpg | Filmato codificato in formato audio MPEG standard |
| .o | File oggetto (output da compilatore, non ancora linkato) |
| .pdf | Documento in formato Adobe PDF (portable document format) |
| .ps | File PostScript |
| .tex | Input per il programma di formattazione TEX |
| .txt | File di testo generico |
| .zip | Archivio compresso |

Figure 1: Esempi di estensioni.

Sequenza Di Record Di Lunghezza Fissa

Un file è una sequenza di record con una struttura interna definita e lunghezza fissa. Il modello storico basato su schede perforate a 80 colonne in mainframe. Letture e scritture avvengono a unità di record, meno comune nei sistemi moderni ma era prevalente nei mainframe passati.

File Come Albero di Record

Il file è organizzato come albero di record, con lunghezze variabili e un campo chiave in posizione fissa. L'organizzazione consente ricerche rapide basate su chiavi specifici. Utilizzato principalmente in sistemi mainframe per elaborazioni dati di carattere commerciale (es. DBMS), diverso dalle sequenze non strutturate di UNIX e Windows.

Molti sistemi operativi supportano diversi tipi di file. UNIX (e di nuovo, macOS e Linux) e Windows, per esempio, hanno file e directory normali. UNIX ha anche file speciali a caratteri o a blocchi.

- **File e Directory Normali:** Sono utilizzati in sistemi come UNIX e Windows. I file normali contengono informazioni utente e sono la forma più comune. Le directory sono file di sistema per mantenere la struttura del file system.
- **File Speciali:** A caratteri usati per modellare porte seriale di I/O come terminali e stampanti. A blocchi usati per modellare dischi.
- **File Normali:** File ASCII composti da righe di testo, visualizzabili e stampabili; variano nella terminazione delle righe. File Binari, non leggibili come testo, hanno una struttura interna conosciuta dai programmi che li utilizzano. Per esempio, file eseguibili o archivi.

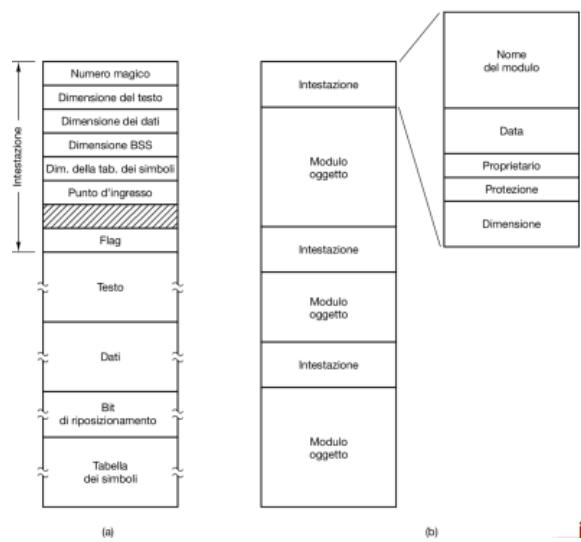


Figure 2: Struttura interna di un file binario.

File Eseguibile - Componenti

- **Intestazione (Header):** Contiene un 'numero magico' per identificare il file come eseguibile (e non eseguire file non "eseguibile"), dimensioni delle parti del file, indirizzo di esecuzione iniziale (punto d'ingresso) e vari flag.
- **Testo e Dati:** Parti effettive del programma, caricare e rilocare in memoria.
- **Tabella dei simboli:** Utilizzata per il debug.

File di Archivio

- **Descrizione:** Raccolta di procedure di libreria (moduli) compilate ma non collegate.
- **Intestazione dei Moduli:** Indicano nome, data di creazione, codice di protezione e dimensione.
- **Carattere Binario:** Stampare questi file produrrebbe caratteri incomprensibili.

Accesso ai File

Nei primi sistemi operativi, c'era un solo metodo di accesso ai file, accesso sequenziale. In questi sistemi, un processo poteva leggere tutti i byte o i record in ordine, a partire dal principio, ma non poteva saltarli né leggerli in ordine sparso. I file sequenziali potevano tuttavia essere riavvolti, in modo da poterli leggere tutte le volte che occorreva.

Successivamente con l'avvento dei dischi è stato introdotto l'accesso causale, che permette la lettura di byte o record in qualsiasi ordine, senza seguire una sequenza. È cruciale per applicazioni come i sistemi di database, dove è necessario accedere rapidamente a record specifici senza attraversare l'intero file. Per specificare dove cominciare a leggere possono essere usati due metodi. Nel primo, ogni operazione read fornisce la posizione del file dalla quale iniziare a leggere. Nel secondo è fornita un'operazione speciale, seek, per impostare la posizione corrente. Dopo una seek il file può essere letto sequenzialmente dalla posizione appena definita come corrente. Quest'ultimo metodo è usato sia in UNIX sia in Windows. Ogni file ha un nome e i propri dati. Tutti i sistemi operativi associano ulteriori informazioni a ciascun file, per esempio la data e l'ora in cui è stato modificato l'ultima volta e la dimensione. Chiameremo attributi (metadati) queste ulteriori voci del file. L'elenco degli attributi cambia in modo considerevole a seconda del sistema.

Gli attributi dei file sono cruciali per:

- la protezione, il controllo dell'accesso
- la gestione efficace dei file nei sistemi operativi

| Attributo | Significato | Attributo | Significato |
|-------------------------|--|---------------------------------------|---|
| Protezione | Chi può accedere al file e in che modalità | Flag temporaneo | 0 per normale; 1 per cancellare il file al termine del processo |
| Password | Password necessaria per accedere al file | Flag di file bloccato | 0 per non bloccato; non zero per bloccato |
| Creatore | ID della persona che ha creato il file | Lunghezza del record | Numero di byte nel record |
| Proprietario | Proprietario attuale | Posizione della chiave | Offset della chiave in ciascun record |
| Flag di sola lettura | 0 per lettura/scrittura; 1 per sola lettura | Lunghezza della chiave | Numero di byte del campo chiave |
| Flag di file nascosto | 0 per normale; 1 per non visualizzare negli elenchi | Data e ora di creazione del file | Data e ora di quando il file è stato creato |
| Flag di file di sistema | 0 per file normali; 1 per file di sistema | Data e ora di ultimo accesso al file | Data e ora di quando è avvenuto l'ultimo accesso al file |
| Flag di file archivio | 0 per già sottoposto a backup; 1 per file di cui fare il backup | Data e ora di ultima modifica al file | Data e ora di quando è avvenuta l'ultima modifica al file |
| Flag ASCII/binario | 0 per file ASCII; 1 per file binari | Dimensione attuale | Numero di byte nel file |
| Flag di accesso casuale | 0 per accesso sequenziale; 1 per accesso casuale | Dimensione massima | Numero di byte di cui può aumentare il file |

Figure 3: Attributi di un file.

Operazioni sui File

Quali operazioni si possono effettuare sui file?

1. **Create:** Creazione di un file senza dati.
2. **Delete:** Eliminazione di un file per liberare spazio sul disco, attraverso una specifica chiamata di sistema.
3. **Open:** Apertura di un file per consentire al sistema di caricare in memoria gli attributi e gli indirizzi del disco.
4. **Close:** Chiusura del file alla termine degli accessi per liberare spazio nelle tabelle interne.
5. **Read:** Lettura dei dati da un file, generalmente dalla posizione corrente, specificando la quantità di dati richiesti e fornendo un buffer per la loro memorizzazione.
6. **Write:** Scrittura di dati nel file, tipicamente alla posizione corrente, può comportare l'ampliamento del file o la sovrascrittura dei dati esistenti.
7. **Append:** Aggiunta dei dati solo alla fine del file, usata in alcuni sistemi operativi come forma limitata di scrittura.
8. **Seek:** Riposizionamento del puntatore del file su una posizione specifica per file ad accesso casuale, permettendo la lettura o la scrittura da quella posizione.
9. **GetAttributes:** Lettura degli attributi del file.
10. **SetAttributes:** Modifica degli attributi di un file da parte dell'utente, come la modalità di protezione o altri flag, dopo la creazione del file.
11. **Rename:** Ridenominazione di un file, utilizzata come alternativa al processo di copia ed eliminazione del file originale, specialmente utile per file di grandi dimensioni.

Directory

Per tener traccia dei file, i file system normalmente hanno directory o cartelle, che sono anch'esse dei file.

Sistemi di directory a livello singolo

La forma più semplice di sistema di directory è di avere una sola directory contenente tutti i file, talvolta chiamata directory principale (root directory). Il vantaggio di questo schema è la semplicità e la capacità di localizzare i file rapidamente: in fin dei conti c'è solo un posto in cui cercare. Oggi, in era moderna, questo tipo di organizzazione viene spesso usato nei dispositivi embedded, come fotocamere digitali o MP3, in tecnologie RFID, come carte di credito, tessere di trasposto.

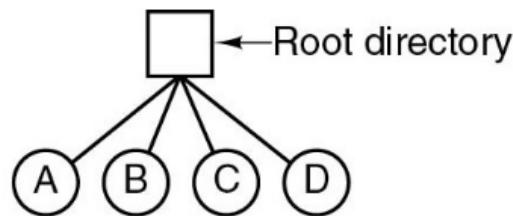


Figure 4: Sistema di directory a livello singolo.

Sistemi a directory gerarchici

Quello che serve è una gerarchia (cioè delle directory ramificate ad albero). Inoltre, se più utenti condividono un file server comune, com'è nel caso di molti network aziendali, ogni utente può avere una directory principale privata per la propria gerarchia. Questo metodo è illustrato nella figura sotto. In questo caso, le directory A, B e C contenute nella directory principale appartengono ciascuna a un utente differente; due utenti hanno creato delle sottodirectory per i progetti su cui stanno lavorando. La capacità di creare un numero arbitrario di sottodirectory fornisce agli utenti un potente strumento di strutturazione per organizzare il lavoro; per questo motivo tutti i file system moderni sono organizzati in questo modo. Vale la pena di ricordare che il file system gerarchico è una delle tante idee sperimentate per la prima volta in Multics negli anni '60.

Quando il file system è organizzato secondo un albero di directory, i nomi dei file vanno specificati in qualche modo. Sono usati comunemente due metodi.

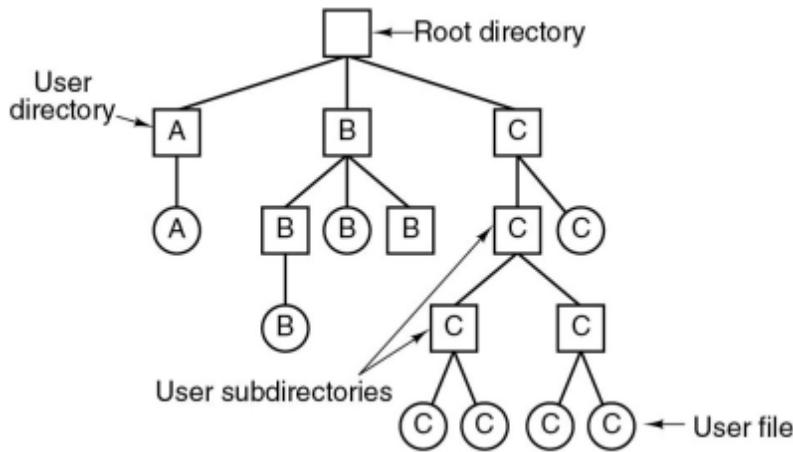


Figure 5: Sistema di directory gerarchico.

Percorso Assoluto

Il primo prevede di assegnare a ogni file un nome di percorso assoluto composto dal percorso che inizia dalla directory principale e arriva al file. Per esempio, il percorso /usr/ast/mailbox significa che la directory principale contiene una sottodirectory usr, che a sua volta contiene una sottodirectory ast, che a sua volta contiene il file mailbox. I nomi di percorso assoluti iniziano dalla directory principale e sono univoci.

Percorso Relativo

L'altro metodo è quello del nome di percorso relativo. È usato congiuntamente al concetto di directory di lavoro, chiamata anche directory corrente. Un utente può designare una directory come directory di lavoro, e in quel caso tutti i nomi di percorso che non cominciano con la directory principale sono considerati relativi alla directory di lavoro. Se l'utente si trova già nella directory /usr/hjb, allora i due comandi sono equivalenti.

Listing 1: Esempio di percorso relativo in UNIX.

```
cp /usr/hjb/mailbox /usr/hjb/mailbox.bak == cp mailbox mailbox.bak
```

In UNIX i componenti del percorso sono divisi tramite /, in Windows il separatore è \ e in MULTICS era >.

Molti sistemi operativi che supportano un sistema di directory gerarchico hanno due voci speciali in ogni directory, “.” e “..”, generalmente dette “punto” e “puntopunto” (“dot” e “dotdot”). “Punto” si riferisce alla directory corrente, “puntopunto” alla directory genitore (la directory precedente), con l'esclusione della directory radice che anche nel caso di “puntopunto” fa riferimento a se stessa.

Operazioni sulle Directory

Quali operazioni si possono effettuare sulle directory?

1. **create:** Creazione di una directory vuota con le voci '.' e '..'
2. **delete:** Eliminazione di una directory, possibile solo se la directory è vuota
3. **opendir:** Apertura di una directory per la lettura del suo contenuto
4. **closedir:** Chiusura di una directory dopo la lettura per liberare risorse
5. **readdir:** Restituisce la prossima voce in una directory aperta senza esporre la struttura interna
6. **rename:** Rinomina una cartella, simile al rinomino di un file
7. **link:** Crea un hard link, collegando un file esistente a un nuovo percorso condividendo l'i-node (blocco di dati presente sul disco)
8. **unlink:** Rimuove una voce di una directory, cancellando il file se è l'unico link

Una variante del concetto di collegare i file è il link simbolico (detto anche collegamento o alias). Invece di avere due nomi che puntano alla stessa struttura di dati interna che rappresenta un file, può essere creato un nome che punta a un piccolo file che simboleggia un altro file. Quando viene usato il primo file, per esempio viene aperto, il file system segue il percorso e trova il nome alla fine. Quindi parte con il processo di ricerca usando il nuovo nome. I link simbolici hanno il vantaggio di poter varcare i confini dei dischi e collegare anche file su computer remoti. La loro implementazione talvolta è tuttavia meno efficiente degli hard link.

Sistemi Operativi

Ionut Zbirciog

5 December 2023

Implementazione del File System

Layout del File System

Il file system è il metodo utilizzato per organizzare e memorizzare dati sui dispositivi di memoria non volatile. Fornisce un modo strutturato per gestire informazioni come file e directory su dispositivi di memoria. Un disco può essere suddiviso in più partizioni, ciascuna con un proprio file system indipendente.

MBR

Nel vecchio stile, il settore 0 del disco è chiamato MBR (Master Boot Record), essenziale per l'avvio del computer. Contiene la tabella delle partizioni (contiene gli indirizzi di inizio e fine di ciascuna partizione) e identifica la partizione attiva da cui avviare il sistema.

Quando si avvia il computer, il BIOS legge ed esegue l'MBR, che localizza la partizione attiva, ne legge il primo blocco, chiamato blocco di boot, e lo esegue. Ogni partizione inizia con un boot block, anche se non contiene un sistema operativo avviabile.

Il layout del file system cambia molto a seconda del file system, ma in generale, è strutturato nel seguente modo:

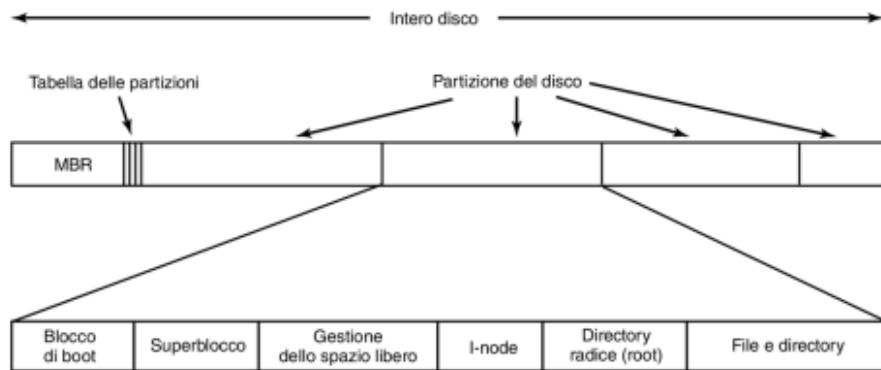


Figure 1: Layout del file system.

- **Superblocco:** Contiene tutti i parametri chiave riguardanti il file system e viene letto in memoria all'avvio del computer. Tipicamente contiene un numero magico per identificare il file system e altre informazioni chiave.
- **Bitmap o Linked List:** Usati per la gestione dello spazio libero.
- **I-node:** Un array di strutture dati, una per file, che contiene tutte le informazioni sui file.
- **Directory radice:** Contiene la cima dell'albero del file system.

UEFI

Avviare il computer con la MBR è lento e limitato a dischi di dimensione fino a 2 TB. Pertanto è stato introdotto UEFI (Unified Extensible Firmware Interface), attualmente il modo più diffuso per avviare il computer. UEFI è veloce, ammette infinite partizioni, supporta diverse architetture e dischi di dimensione fino a 8 ZiB.

UEFI non si basa sul MBR ma cerca la tabella delle partizioni nel secondo blocco, riservando il primo blocco per il software che si aspetta di trovare un MBR.

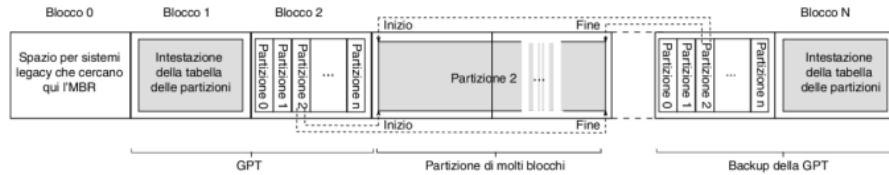


Figure 2: UEFI con GPT.

UEFI utilizza la GPT (GUID Partition Table), una struttura flessibile e dinamica che contiene informazioni sulla posizione delle partizioni sul disco. Nell'ultimo blocco conserva un backup della GPT. Una GPT contiene l'inizio e la fine di ogni partizione, e una volta trovata la GPT, il firmware ha funzionalità per leggere file system di tipi specifici.

Implementazione dei File

È importante avere una buona implementazione dei file per assicurare l'integrità, l'accesso efficiente e la gestione dello spazio sul disco.

Allocazione Continua

Lo schema di allocazione più semplice è memorizzare ciascun file come una sequenza contigua di blocchi sul disco.

Vantaggi:

- Semplice da implementare.
- Prestazioni di lettura eccellenti.

Svantaggi:

- Frammentazione del disco.
- Difficoltà nell'aggiungere nuovi file in un disco frammentato.
- Necessità di compattare il disco, operazione costosa.

Allocazione a Liste Concatenate

Il secondo metodo per memorizzare i file è configurare ciascuno come una lista concatenata di blocchi sul disco. La prima parte di ciascun blocco è usata come puntatore al successivo, il resto del blocco è per i dati.

Vantaggi:

- Utilizza in modo efficiente tutti i blocchi del disco.
- Minimizza la frammentazione esterna.

Svantaggi:

- Accesso casuale estremamente lento.
- Dimensioni dei blocchi non più una potenza di due, riducendo l'efficienza.

Allocazione a Liste Concatenate con FAT

Gli svantaggi dell'allocazione a liste concatenate possono essere eliminati spostando i puntatori in una tabella di memoria FAT (File Allocation Table). Ogni blocco del disco è rappresentato come una voce nella FAT, in memoria RAM.

Vantaggi:

- Accesso casuale semplificato, poiché i puntatori sono in RAM.

Svantaggi:

- La tabella deve rimanere sempre in memoria.
- Occupa considerevole quantità di RAM, specialmente su dischi grandi.



Figure 3: Allocazione a Liste Concatenate con FAT.

Domande

Quanti indirizzi si possono mettere su un blocco da 4 KB, se ogni numero/indirizzo è rappresentato a 32 bit (4 byte)?

$$\frac{2^{12}}{2^2} = 2^{10} = 1024 \text{ indirizzi}$$

Qual è il file più grande che si può indicizzare nei seguenti sistemi di file:

- **FAT12:** 32 MB
- **FAT16:** 2 GB
- **FAT32:** 4 GB

I-node

Gli i-node o index-node sono una struttura dati che elenca gli attributi (esclusi nome e contenuto) come permessi, proprietario, timestamp e gli indirizzi dei blocchi dei file. Ogni file e directory è rappresentato da un I-node univoco, indicizzato in una tabella di I-node.

Dato un i-node, è possibile trovare tutti i blocchi di quel file. Un grande vantaggio è che solo gli i-node dei file aperti sono mantenuti in memoria, riducendo significativamente l'utilizzo della memoria. L'array degli i-node in memoria è proporzionale al numero di file aperti, non alla dimensione del disco.

Gli i-node hanno uno spazio limitato per gli indirizzi, per file che superano il limite, uno degli indirizzi nell'i-node punta a un blocco contenente ulteriori indirizzi di blocchi dati.

Gli i-node sono un concetto fondamentale in UNIX e nei suoi file system derivati. NTFS, il file system di Windows, utilizza una struttura simile con i-node più grandi che possono contenere file di piccole dimensioni all'interno dell'i-node stesso.

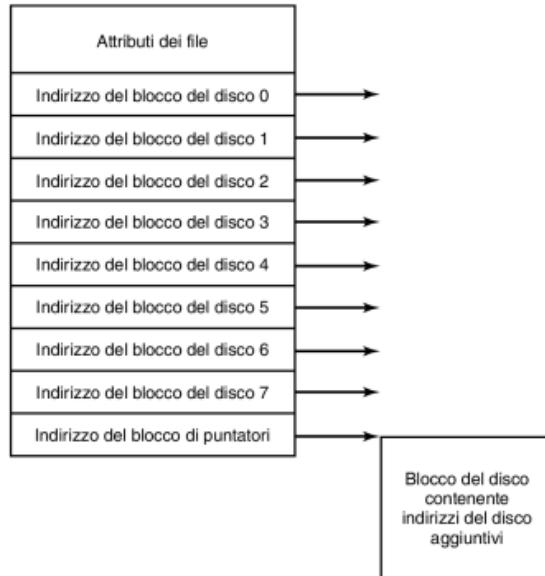


Figure 4: Struttura degli I-node.

Implementazione delle Directory

Implementazione Directory

Il ruolo principale delle directory è di mappare il nome ASCII del file sulle informazioni necessarie per localizzare i dati sul disco. A seconda del sistema, questa informazione può essere l'indirizzo del disco dell'intero file (allocazione contigua), il numero del primo blocco (linked list), o il numero dell'i-node.

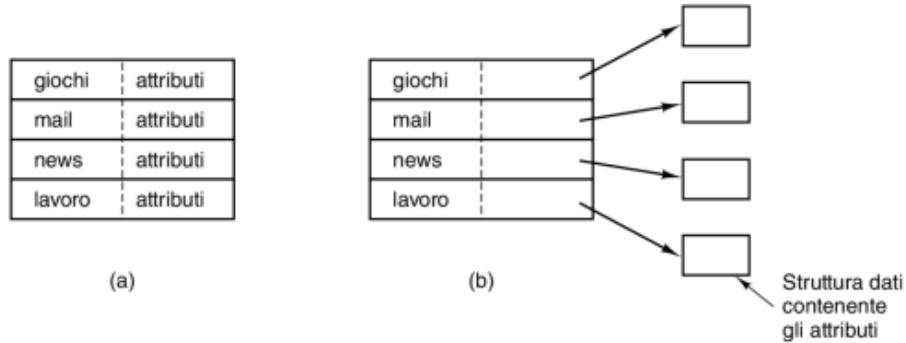


Figure 5: Struttura delle directory.

I moderni file-system supportano nomi di file variabili da 1 a 255 caratteri. Per fare ciò, le directory si possono strutturare in 2 modi:

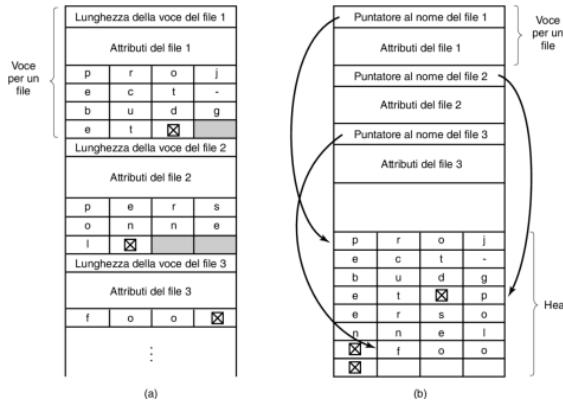


Figure 6: (a) Directory con header di lunghezza fissa. (b) Uso di heap per i nomi dei file.

(a) Ciascuna directory contiene l'header di lunghezza fissa seguito dal nome del file. Ogni nome di file termina con un carattere speciale. Per consentire a ciascuna voce di directory di iniziare alla fine della parola, ogni nome di file è riempito fino ad arrivare a un numero intero di parole. Lo svantaggio è che quando il file viene cancellato, nella directory è introdotto un vuoto di ampiezza variabile, che potrebbe non bastare a contenere il file da inserire successivamente.

(b) Uso di heap per tenere i nomi dei file. Questo metodo ha il vantaggio che quando viene rimossa una voce, il successivo file da inserire ci starà sempre. Una piccolissima conquista in questo caso è che non è più necessario che i nomi dei file iniziino ai confini delle parole, quindi non sono necessari caratteri di riempimento.

Inizialmente, i file in una directory venivano cercati linearmente dall'inizio alla fine. Questo metodo può diventare lento in directory con un gran numero di file. Si è passato dunque all'utilizzo delle Hash Table, accelerando il processo di ricerca. Il nome di un file è sottoposto a hashing per generare un indice nell'intervallo da 0 a $n - 1$. La voce corrispondente nella tabella di hash indica il punto di partenza per la ricerca del file. Per i file che condividono lo stesso hash, viene creata una lista concatenata. Un modo diverso per velocizzare la ricerca in grandi directory è salvare nella cache il risultato delle ricerche. Prima di avviare la ricerca, si verifica se quel nome di file si trova nella cache. Se si trova, può essere immediatamente individuato; altrimenti, si segue la ricerca all'interno della directory.

File Condivisi e Link nel File System

I file condivisi sono essenziali in ambienti collaborativi per permettere a più utenti di lavorare sugli stessi file.

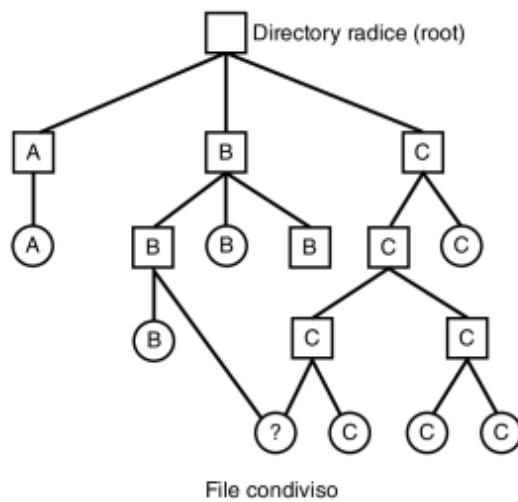


Figure 7: (a) Hard Link. (b) Soft Link.

- **Hard Link:** Puntano direttamente all'i-node di un file condiviso. Un file con hard link viene rimosso solo quando non ci sono più riferimenti ad esso. Sono efficienti in termini di spazio, usano un solo i-node indipendentemente dal numero di link. Il file permane fino all'eliminazione dell'ultimo link, potenzialmente causando confusione sulla proprietà del file.
- **Soft Link:** Puntano al nome di un file piuttosto che all'i-node. Sono più flessibili, possono riferirsi a nomi di file oltre i confini del file system e su macchine remote, ma sono meno

efficienti in termini di spazio in quanto richiedono un i-node per ogni link. I soft link diventano invalidi alla rimozione del file originale. Il sistema operativo impiega più tempo nella risoluzione del percorso rispetto agli hard link.

Entrambe le modalità, però, hanno un problema comune, ovvero che i file con più percorsi possono essere processati più volte da programmi di backup o di ricerca. Per esempio, c'è il rischio di duplicazione dei file su un'unità di backup.

Gestione Dello Spazio sul Disco

Dimensione dei Blocchi

Generalmente i file sono memorizzati su disco, tramite l'allocazione contigua, portando a maggiori spostamenti sul disco se le loro dimensioni aumentano, oppure suddividendo il file in blocchi più piccoli consentendo maggiore flessibilità e un migliore utilizzo dello spazio su disco. La dimensione comune di 4KB per blocco è un compromesso tra lo spazio su disco e le prestazioni di trasferimento dei dati.

Perché 4KB?

- **Prestazioni di Trasferimento Dati:** I dischi magnetici con blocchi più grandi consentono il trasferimento di più dati per operazione di lettura/scrittura. Ma blocchi grandi portano a spreco di memoria.
- **Efficienza dello Spazio:** Blocchi piccoli minimizzano lo spreco di spazio con file piccoli. Ma significa distribuire la maggior parte dei file su più blocchi e incorrere in più ricerche e ritardi per leggerli. L'efficienza dello spazio diminuisce con l'aumento della dimensione dei blocchi.

Gestione dei Blocchi Liberi

1. Linked List

Si usa una lista concatenata, in cui si vanno a mettere solo i blocchi liberi. Per ospitare la lista si usano gli stessi blocchi del disco. Ogni blocco contiene numeri di blocchi del disco liberi. Richiede meno spazio solo se il disco è quasi pieno. Per ottimizzare la lista, si possono tracciare i blocchi liberi consecutivi. A ciascun blocco può essere associato un conteggio a 8, 16, 32 bit che rappresenta il numero di blocchi liberi consecutivi. Nell'ipotesi migliore, un disco fondamentalmente vuoto è rappresentato da due numeri, l'indirizzo del primo blocco libero, seguito dal conteggio dei blocchi liberi. Questo metodo è migliore per dischi quasi vuoti, ma meno efficiente per dischi molto frammentati.

La gestione dei blocchi liberi può utilizzare una lista concatenata di puntatori, nota come "free list". Solo un blocco di puntatori è mantenuto in memoria contemporaneamente, ottimizzando così l'utilizzo della memoria. Quando si crea un file, i blocchi necessari vengono allocati dai puntatori disponibili nel blocco in memoria. Questo metodo evita I/O su disco inutili mantenendo una lista di blocchi liberi direttamente accessibili in memoria. Al riempimento del blocco di puntatori in memoria, un nuovo blocco viene letto da disco per proseguire con le operazioni. La presenza di file temporanei può portare a frequenti operazioni di I/O su disco se il blocco di puntatori in memoria è quasi pieno. Una strategia alternativa prevede di dividere il blocco pieno di puntatori per gestire meglio i blocchi liberi senza I/O su disco.

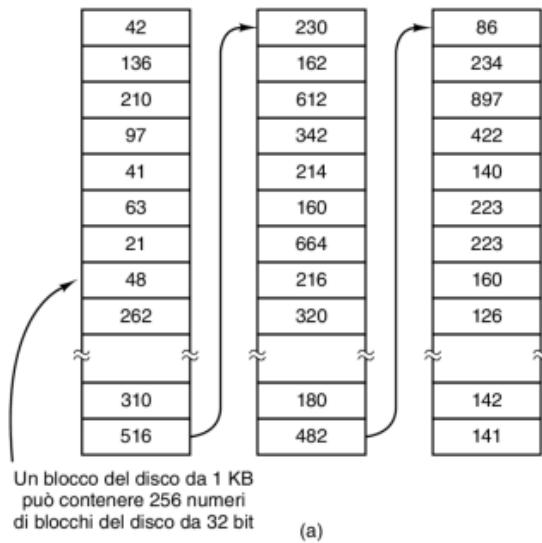


Figure 8: Linked List.

2. Bitmap

Un disco con n blocchi richiede una bitmap con n bit. I blocchi liberi sono indicati dal valore 1 nella mappa, quelli allocati dallo 0. Per esempio, per un disco da 1TB serve una mappa da 1 miliardo di bit, il che richiede 130,000 blocchi da 1 KB per eseguire la memorizzazione. È più efficiente rispetto alla linked list, tranne in dischi quasi pieni.

| |
|------------------|
| 1001101101101100 |
| 011011011110111 |
| 101011010110110 |
| 0110110110111011 |
| 1110111011101111 |
| 1101101010001111 |
| 0000111011010111 |
| 1011101101101111 |
| 1100100011101111 |
| ~ ~ |
| 0111011101110111 |
| 1101111101110111 |

Una bitmap

(b)

Figure 9: Struttura di una bitmap.

Quote del Disco

Per impedire che gli utenti occupino troppo spazio su disco, i sistemi operativi multiutente forniscono spesso un meccanismo per impostare le quote del disco. L'idea è che l'amministratore di sistema assegna a ciascun utente un numero massimo di file e blocchi, e che il sistema operativo si accerti che gli utenti non superino la loro quota. Ogni apertura di file coinvolge il controllo degli attributi e degli indirizzi sul disco. Gli attributi includono l'identificatore del proprietario del file. In una tabella dei file aperti, viene contabilizzata la quota di ciascun utente. Ci sono 2 limiti, soft e hard. Il limite soft può essere temporaneamente superato, mentre il limite hard, no.

Sistemi Operativi

Ionut Zbirciog

7 December 2023

Performance del File System

C'è un gap di velocità di accesso troppo grande tra RAM e memoria volatile, negli ultimi anni questo gap è stato minimizzato con l'invenzione degli SSD. Per leggere una parola a 32 bit, la RAM ci mette circa 10ns mentre un disco magnetico ci mette circa 10ms. Pertanto, bisogna progettare un file system con diverse ottimizzazioni per migliorare le prestazioni, considerando le significative differenze nel tempo di accesso e riducendo al minimo i numeri di accesso al disco/SSD.

Ci sono due tecniche principali per ottimizzare il file system.

Uso della Cache

Utilizzata per ridurre i tempi di accesso al disco mantenendo i blocchi più usati in memoria. La cache si trova all'interno della memoria RAM.

Concetti di Caching

- **Buffer Cache:** Memorizza i blocchi del disco in RAM per ridurre gli accessi al disco.
- **Page Cache:** Memorizza le pagine del filesystem virtuale (VFS) in RAM prima di passare al driver del dispositivo. Sono struttura di intermezzo tra il virtual file system e il vero file system.

Le cache devono essere ottimizzate in modo tale che non ci siano file duplicati, perché può succedere che buffer cache e page cache spesso contengono gli stessi dati, per esempio file "mappati in memoria".

In poche parole, i file sono nella cache delle pagine e i blocchi del disco sono nella cache buffer. I sistemi operativi possono combinare buffer cache e page cache per un uso più efficiente della memoria e una riduzione degli accessi al disco.

Implementazione della Cache

Per la gestione della cache si può usare il seguente algoritmo: si controllano tutte le richieste di lettura per verificare se il blocco necessario sia nella cache. Se lo è, la richiesta di lettura è soddisfatta senza accedere al disco. Se non lo è, per prima cosa viene letto da disco e portato nella cache, quindi copiato ovunque ve ne sia bisogno. Le successive richieste per lo stesso blocco potranno essere soddisfatte dalla cache.

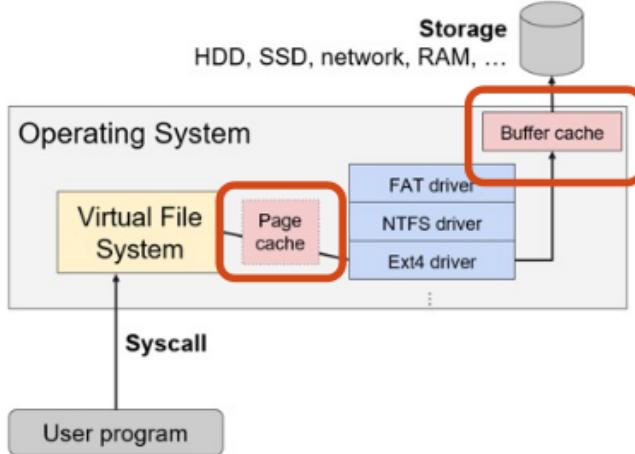


Figure 1: Struttura della cache.

Generalmente nella cache vi sono molti blocchi e serve un metodo rapido per determinare se un blocco sia o meno nella cache. Viene quindi utilizzata una hash table per velocizzare la ricerca di un blocco all'interno di un blocco.

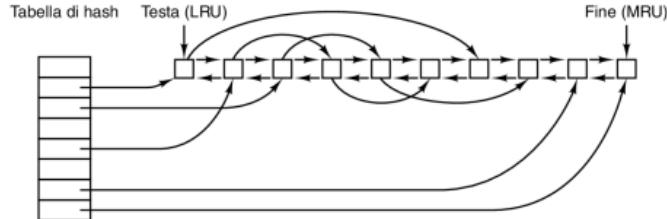


Figure 2: Struttura di una hash table nella cache.

Quando un blocco deve essere caricato in una cache piena, ne deve essere rimosso qualcuno. Questa situazione è molto simile alla paginazione e sono applicabili tutti gli abituali algoritmi di sostituzione delle pagine, come FIFO, seconda chance ed LRU. Una gradevole differenza fra l'uso della paginazione e della cache sta nel fatto che i riferimenti alla cache sono relativamente poco frequenti, così è possibile mantenere tutti i blocchi nell'esatto ordine LRU con le liste concatenate.

C'è un unico problema però: LRU può portare a incosistenza in caso di crash, specialmente per blocchi critici come gli i-node. Per esempio, un blocco di i-node viene letto nella cache e modificato, ma non riscritto su disco, un crash del file system lo lascerebbe in uno stato incoerente. Se il blocco di i-node viene posto alla fine della catena LRU, può volerci parecchio tempo prima che raggiunga la testa e sia riscritto sul disco. Per risolvere questo problema si può pensare a modificare

LRU, dividendo i blocchi per categorie basate sulla loro importanza. Se un blocco è fondamentale per la coerenza del file system, questo blocco dovrebbe essere scritto immediatamente sul disco, indipendentemente dalla sua posizione nella LRU.

Oltre a queste tecniche, UNIX mette a disposizione una chiamata di sistema 'sync' che forza la scrittura di tutti blocchi modificati. Al momento dell'avvio del sistema, viene eseguito un processo in background che ad ogni 30 secondi scrive tutti i blocchi modificati su disco. Ad oggi esiste una chiamata simile anche in Windows, ma prima veniva usata un'altra tecnica detta write-through, ovvero il sistema scriveva su disco ogni blocco che veniva modificato appena veniva scritto in cache. Una conseguenza di tale differenza nella strategia di caching è che la rimozione di un disco da un sistema UNIX senza fare la sync (per questo è opportuno prima di togliere un disco, fare umount) provoca quasi certamente una perdita di dati, e spesso intacca anche il file system. Con le cache write-through il problema non si pone.

Concettualmente cache page e cache buffer sono diversi nel senso che la cache delle pagine contiene le pagine dei file per ottimizzare l'I/O, mentre la cache buffer contiene semplicemente blocchi del disco. La cache buffer, nata prima della cache delle pagine, in realtà si comporta un po' come un disco, tranne che letture e scritture avvengono nella memoria. Il motivo per cui è stata aggiunta una cache delle pagine è che sembrava una buona idea portare la cache più in alto nello stack, in modo che le richieste di file potessero essere soddisfatte senza passare per il codice del file system e tutte le sue complessità. In altre parole: i file sono nella cache delle pagine e i blocchi del disco nella cache buffer. Alcuni sistemi operativi combinano cache buffer con cache page per una gestione efficiente dei dati. Inoltre, supportano i file mappati in memoria, trattando allo stesso modo pagine e blocchi del disco, in un'unica cache.

Allocazione dei Blocchi e Read Ahead

Un'altra tecnica importante per i dischi magnetici è quella di ridurre la quantità di movimenti del braccio mettendo vicini, preferibilmente nello stesso cilindro, i blocchi ai quali è probabile si acceda in sequenza. Quando viene scritto un file di output, il file system deve allocare i blocchi uno alla volta, a richiesta. Se i blocchi liberi sono registrati in una bitmap e l'intera bitmap è nella memoria principale, è abbastanza semplice scegliere un blocco libero che sia il più vicino possibile al blocco precedente. Se invece si usa la lista dei blocchi liberi, parte della quale sta su disco, è molto più difficile allocare blocchi vicini l'uno all'altro.

Altro collo di bottiglia delle prestazioni nei sistemi che usano gli i-node, o qualcosa di analogo, è che la lettura di un file, anche piccolo, richiede due accessi al disco: uno per l'i-node e uno per il blocco.

- a) Posizionamento tradizionale degli I-node: I-node vicino all'inizio del disco, risultando in ricerche più lunghe.
- b) Centrare gli I-node: Posizionare gli I-node nel mezzo del disco per dimezzare il tempo di ricerca. Dividere il disco in gruppi di cilindri con I-node, blocchi e lista dei blocchi liberi per ciascun gruppo.

Ovviamente, i movimenti del braccio del disco e il tempo di rotazione sono importanti solamente se il disco è magnetico; non sono rilevanti per gli SSD, che non hanno parti mobili. Per questi dischi, costruiti con la tecnologia delle schede flash, l'accesso casuale (in lettura) è rapido quanto quello sequenziale.

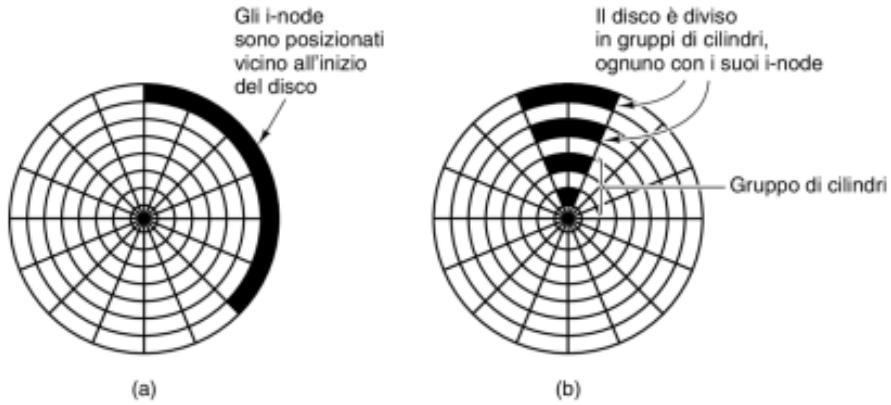


Figure 3: Posizionamento degli I-node.

Deframmentazione dei dischi

Con il passare del tempo, i file vengono creati ed eliminati e di norma il disco diventa molto frammentato, con file e buchi sparsi ovunque. Di conseguenza, quando viene creato un nuovo file i blocchi utilizzati potrebbero essere sparpagliati per tutto il disco, ottenendo basse prestazioni.

Le prestazioni possono essere ripristinate spostando i file in modo che siano contigui e mettendo tutto (o la maggior parte) dello spazio libero in una o più grandi zone continue su disco. Windows ha un programma, defrag, che fa esattamente questo. Gli utenti di Windows dovrebbero eseguirlo con regolarità, ma non sugli SSD.

I file system di Linux (specialmente ext3, ext4, btrfs) gestiscono meglio la frammentazione, riducendo la necessità di deframmentazione manuale. Ext4 introduce la preallocazione dei blocchi. Quando si scrive un file, Ext4 preallocata un gruppo di blocchi contigui, piuttosto che uno alla volta, riducendo la frammentazione per i file in espansione.

Compreensione e Deduplicazione

Alcuni file system come NTFS, Btrfs possono comprimere dati automaticamente e scrivere sul disco i dati compressi per risparmiare spazio. Oltre a eliminare la ridondanza entro un singolo file, alcuni file system eliminano anche la ridondanza in tutti i file. Nei sistemi che memorizzano dati di molti utenti, ad esempio in un ambiente cloud o server, è facile trovare file che contengono gli stessi dati quando più utenti salvano gli stessi documenti, file binari o video. Anziché salvare più volte gli stessi dati, alcuni file system implementano la deduplicazione per eliminare i duplicati.

La deduplicazione può essere eseguita inline o post-process. Con la deduplicazione inline, il file system calcola uno hash per ogni chunk (“pezzo” di memoria) che sta per scrivere e lo confronta con gli hash dei chunk esistenti. Se il chunk è già presente, eviterà di scrivere i dati e aggiungerà invece un riferimento (hard link) al chunk esistente. Naturalmente i calcoli aggiuntivi richiedono tempo e rallentano la scrittura. La deduplicazione post-process, invece, scrive sempre i dati ed esegue l'hashing e i confronti in background, senza rallentare le operazioni di elaborazione dei file.

Affidabilità del File System

Minacce alla Affidabilità del File System

- **Guasti del Disco:** blocchi danneggiati o settori illeggibili che possono corrompere dati, o errori su intero disco, rendendo il disco inutilizzabile.
- **Interruzioni di Energia:** causano incongruenze nei dati o nei metadati sul disco.
- **Bug del Software:** errori di programmazione portano alla scrittura di dati errati/corrotti.
- **Errori Umani:** ad esempio, "rm *.o" vs "rm * .o"; il primo cancella tutti i file con estensione '.o', mentre il secondo cancella tutti i file (*) e poi, se ci sono ancora tutti i file che finiscono con '.o'.
- **Perdita o Furto del Computer:** rischi di accesso ai dati da parte di persone non autorizzate in caso di furto o smarrimento del dispositivo.
- **Malware/Ransomware:** virus o altri software dannosi che possono infettare, criptare o distruggere dati.

Backup

I backup su disco sono generalmente effettuati per affrontare uno dei due potenziali problemi:

1. **Recupero da un Disastro:** crash del disco, incendio, catastrofe naturale.
2. **Recupero dalla Stupidità:** eliminazione accidentale di file; ad esempio, in Windows esiste il cestino, una cartella in cui vanno i file 'eliminati', in modo poi da poterli recuperare.

Ci sono cinque punti fondamentali di cui tener conto.

1. Un backup impiega molto tempo e occupa molto spazio, per questo è importante farlo in modo efficiente e comodo. Di quali file bisogna fare il backup? Di tutto il file system o solo di alcune directory?
2. È uno spreco rifare il backup di file che non sono cambiati dall'esecuzione dell'ultimo backup, e questo ci porta al concetto di backup incrementale. La forma incrementale più semplice consiste nel fare periodicamente un backup completo e eseguire un backup giornaliero solo dei file modificati dopo l'ultimo backup completo.
3. Poiché generalmente sono memorizzate grandi quantità di dati, potrebbe essere utile compressere i dati prima di scriverli sul supporto di backup.
4. È difficile eseguire il backup di un file system attivo. Sono stati definiti algoritmi che creano delle istantanee (snapshot) dello stato del file system.
5. I backup devono essere sempre tenuti al di fuori del luogo dove risiedono i computer, introducendo ulteriori rischi per la sicurezza.

Strategie di Backup su Disco

- **Backup Fisico:** Copia sequenziale di tutti i blocchi del disco, semplice e veloce, ma ha difficoltà nel saltare le directory specifiche o nel fare backup incrementali. Non è possibile ripristinare file individuali senza un intero ripristino del sistema.
- **Backup Logico:** Seleziona e copia solo i file e le directory specifici, ideale per backup incrementali o completi. Permette il ripristino o il trasferimento dell'intero file system su un nuovo computer.

Algoritmo di Backup Logico

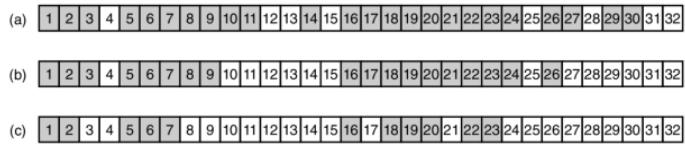


Figure 4: Algoritmo di backup logico.

L'algoritmo consiste di 4 fasi:

- **Fase 1. Rilevamento delle modifiche:** Inizia dalla directory radice, esamina tutte le voci e contrassegna nella bitmap gli I-node di file e directory modificati.
- **Fase 2. Pulizia della Bitmap:** Deseleziona le directory che non contengono né file modificati né sottodirectory modificate.
- **Fase 3. Backup delle Directory:** Backup delle directory contrassegnate, con i loro attributi.
- **Fase 4. Backup dei File:** Backup dei file contrassegnati, sempre con i relativi attributi.

Per ripristinare un file system con un backup, si crea un nuovo file system vuoto, poi viene ripristinato il backup completo, seguito dai backup incrementali.

- Dato che la lista dei blocchi liberi non è un file, non è oggetto di backup e perciò deve essere ricostruita da zero alla fine del ripristino di tutti i backup.
- Se un file è collegato a una o più directory tramite link, è importante che quel file sia ripristinato solo una volta e che tutte le directory che dovrebbero puntare a esso lo facciano.
- Un'ulteriore questione è il fatto che i file UNIX possono contenere dei buchi. È consentito aprire un file, scriverci pochi byte, quindi spostarsi a un offset distante e scrivervi altri byte. I blocchi nel mezzo non sono parte del file, non dovrebbero essere salvati nel backup né ripristinati.
- File speciali come "pipe" e altri file simili non andrebbero mai salvati nel backup, indipendentemente dalla directory in cui si possano trovare.

Coerenza

La coerenza del file system è cruciale per mantenere l'integrità dei dati. Problemi di incoerenza possono sorgere a seguito di crash durante la scrittura dei blocchi. In UNIX si utilizza `fsck`, e in Windows `sfc` per verificare la coerenza all'avvio dopo un crash.

Controllo dei blocchi

Costruzione di due tabelle con contatori per ogni blocco (per file e blocchi liberi). Analisi di tutti gli I-node e verifica della presenza dei blocchi in file e nella lista dei blocchi liberi. Dopo questo controllo si possono avere 3 riscontri possibili:

1. Blocchi mancanti: non presenti in nessuna delle due tabelle.
2. Blocchi duplicati nella lista dei blocchi liberi.
3. Blocchi di dati presenti in più file.

Per correggerli si possono fare due azioni:

1. Aggiunta dei blocchi mancanti ai blocchi liberi.
2. Assegnazione di blocchi duplicati a file diversi.

File System con Journaling

Registra anticipatamente le operazioni da eseguire in un log, per garantire la coerenza in caso di crash. Ampiamente usato in NTFS, ext4, macOS. Un journal in un file system è come un registro che tiene traccia delle modifiche che verranno apportate al file system prima che esse avvengano effettivamente.

Funzionamento:

- **Fase di Registrazione:** prima di eseguire qualsiasi modifica, il file system scrive un record nel journal, descrivendo l'operazione che verrà eseguita.
- **Fase di Esecuzione:** dopo aver registrato l'operazione, il file system procede con la modifica effettiva dei dati sul disco.
- **Fase di Conferma:** una volta completata l'operazione, il file system aggiorna il journal per indicare che l'azione è stata completata con successo.

Se si verifica un crash del sistema prima che un'operazione sia completata, al riavvio successivo il file system constata il journal. Se trova operazioni registrate ma non confermate procede a completarle, in questo modo il file system non verrà mai lasciato in uno stato incoerente.

Vantaggi:

- **Integrità dei Dati:** il journaling riduce la possibilità di corruzione del file system in caso di crash inaspettato, assicurando che tutte le operazioni siano completate o nessuna.
- **Recupero Rapido:** Riduce significativamente il tempo di recupero dopo un crash.

Eliminazione sicura e cifratura del disco

La cancellazione standard non rimuove fisicamente i dati dal disco, lasciandoli vulnerabili agli attacchi. L'eliminazione sicura richiede la distruzione fisica o la sovrascrittura approfondita dei dati. Non basta sovrascrivere con zero a causa dei residui magnetici che possono essere recuperati con tecniche avanzate. È consigliato inserire sequenze di zeri e numeri casuali, ripetendo l'operazione almeno 3-7 volte, sconsigliato su SSD.

La soluzione più efficace per proteggere i dati è cifrare l'intero disco con algoritmi robusti come l'AES.

File System Virtuali

I sistemi operativi moderni gestiscono diversi file system simultaneamente. Windows utilizza lettere come C:, D:, per gestire file system differenti. I sistemi UNIX tentano di integrare più file system in una singola struttura gerarchica.

Il VFS è una struttura che permette di integrare vari file system in una struttura unificata. Si basa su un livello di codice comune che interagisce con i file system reali sottostanti. L'interfaccia superiore interagisce con le chiamate di sistema POSIX di processi utente, mentre l'interfaccia inferiore è composta da decine di funzioni che il VFS può inviare ai file system sottostanti.

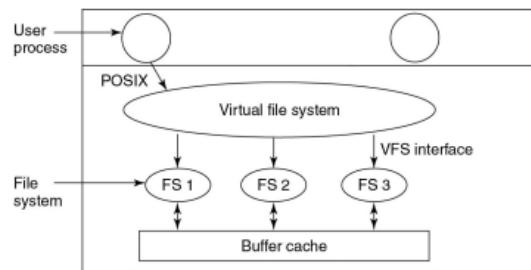


Figure 5: Struttura del File System Virtuale (VFS)

- **Superblock nel VFS:** rappresenta il descrittore di alto livello di un file system specifico nel VFS. Contiene informazioni cruciali sul file system ed è usato per identificare e interagire con il file system sottostante.
- **V-node nel VFS:** astrazione di un file individuale all'interno del VFS, rappresentando un nodo nel file system virtuale. Contiene metadati come permessi, proprietà, dimensione del file. Il VFS sfrutta i v-node per fornire.
- **Directory nel VFS:** struttura che gestisce l'organizzazione e il mapping dei file e delle sottodirectory all'interno del VFS. Permette al VFS di mappare i nomi dei file ai loro v-node corrispondenti, indipendentemente dal file system in cui si trovano.

Al momento della registrazione di un nuovo file system all'interno del VFS, il file system mette a disposizione un vettore di funzioni per il VFS. Inoltre, al montaggio, il file system mette a

disposizione tutte le informazioni necessarie, ad esempio il superblock. La gestione delle richieste di I/O nei processi utente avviene tramite i v-node e tabelle dei descrittori dei file. Aggiungere un nuovo file system diventa relativamente semplice, i progettisti devono soltanto fornire le funzioni che rispettino l'interfaccia VFS.

RAID

RAID, acronimo di Redundant Array Of Inexpensive (Independent) Disks, è un meccanismo per migliorare le prestazioni e l'affidabilità della memoria non volatile.

- RAID Level 0: +storage Descrizione dei dati in strip su dischi multipli, migliora le prestazioni con richieste grandi. Nessuna ridondanza.
- RAID Level 1: +redundancy Duplicazione dei dischi per tolleranza agli errori. Prestazioni di lettura migliori, scrittura simile a un'unità singola.
- RAID Level 2: +storage Level 2 basato su parole o byte con codice di Hamming.
- RAID Level 3: +redundancy, +storage Level 3 usa un singolo bit di parità per parola, richiede sincronizzazione delle unità.
- RAID Level 4: +redundancy, +storage Utilizzo di strip con un'unità extra per la parità.
- RAID Level 5: +redundancy, +storage Distribuisce bit di parità in modo uniforme su tutte le unità.
- RAID Level 6: +redundancy, +storage Simile a Level 5 ma con un blocco di parità aggiuntivo. Maggiore affidabilità e tolleranza degli errori.
- RAID 0 + 1: duplicazione ridondante secondo lo schema di RAID 1 di dischi associati in schema RAID 0

Esempio RAID 5

RAID 5 con 3 dischi: Disco A, Disco B, Disco C. Ogni disco contiene una strip di dati e una strip di parità viene calcolata usando l'operazione XOR bit a bit dei dati da Disco A a Disco B, e poi memorizza sul Disco C.

- Disco A: 1011
- Disco B: 1100
- XOR bit a bit tra A e B
- Disco C: 0111

In caso di guasto di un disco, ad esempio il disco B, possiamo ricostruire i suoi dati facendo lo XOR tra il disco A ancora funzionante e il disco C.

Sistemi Operativi

Ionut Zbirciog

14 December 2023

Principi dell'Hardware di I/O

Dispositivi di I/O

I dispositivi di I/O si possono suddividere in:

- **Dispositivi a blocchi:** Archiviano informazioni in blocchi di dimensioni fisse, ognuno con il proprio indirizzo. Tutti i trasferimenti sono in unità di uno o più blocchi interi. Caratteristica principale: ogni blocco è indipendente dagli altri. Esempi di dispositivi a blocchi sono dischi fissi magnetici, SSD, ecc.
- **Dispositivi a caratteri:** Rilasciano o accettano un flusso di caratteri senza alcuna struttura a blocchi. Non sono indirizzabili e non hanno operazioni di ricerca. Esempi di dispositivi a caratteri sono stampanti, mouse, tastiere, ecc.

Questa classificazione, tuttavia, non include tutti i dispositivi di I/O, come ad esempio clock, schermi o touchscreen.

| Dispositivo | Velocità di trasferimento |
|---------------------------------|---------------------------|
| Tastiera | 10 byte/s |
| Mouse | 100 byte/s |
| Modem a 56 K | 7 KB/s |
| Bluetooth 5 BLE | 256 KB/s |
| Scanner a 300 dpi | 1 MB/s |
| Videocamera digitale | 3,5 MB/s |
| Wireless 802.11n | 37,5 MB/s |
| USB 2.0 | 60 MB/s |
| Disco Blu-ray 16x | 72 MB/s |
| Gigabit Ethernet | 125 MB/s |
| Disco fisso SATA 3 | 600 MB/s |
| USB 3.0 | 625 MB/s |
| Bus PCIe 3.0 single lane | 985 MB/s |
| Wireless 802.11ax | 1,28 GB/s |
| SSD NVME PCIe Gen 3.0 (lettura) | 3,5 GB/s |
| USB 4.0 | 5 GB/s |
| PCI Express 6.0 | 128 GB/s |

Figure 1: Dispositivi di I/O

Controller dei Dispositivi

I dispositivi di I/O sono composti da una parte elettronica chiamata *controller del dispositivo* e da una parte meccanica, ovvero il dispositivo stesso. Il controller è presente nei computer come un chip o una scheda a circuiti stampati. Molti controller possono gestire diversi dispositivi identici. Le interfacce fra il controller e il dispositivo possono essere standardizzate come ANSI, IEEE, ISO o de facto, come SATA, SCSI, USB o Thunderbolt.

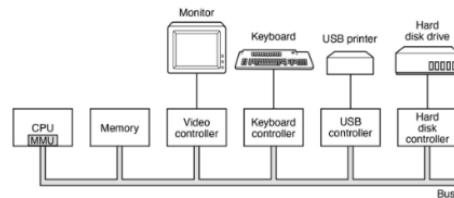


Figure 2: Dispositivi e vari controller

L'interfaccia tra il controller e il dispositivo è di livello molto basso; il compito del controller è convertire il flusso seriale di bit in blocchi di byte, correggere errori e trasferire in memoria centrale. Senza il controller, il programmatore dovrebbe gestire dettagli complessi, come la modulazione di ciascun pixel. Il controller viene inizializzato dal sistema operativo con parametri essenziali, poi viene delegato alla gestione dei dettagli complessi.

Dalla Porta parallela alla Porta USB

Porta Parallela: Tipo di interfaccia di comunicazione tra computer e dispositivi. Trasmette dati multi-bit simultaneamente su più canali. Comunemente dotata di 25 o 36 pin.

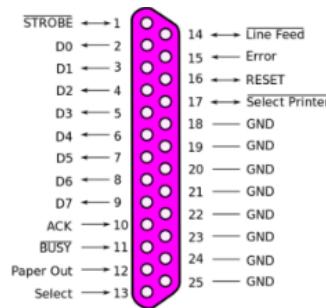


Figure 3: Porta Parallela

- Pin 1-8: Trasferimento di dati
- Pin 9-16: Controlli e status
- Pin 17-25: Masse e alimentazione

Porta USB: Universal Serial Bus è un’interfaccia standardizzata per la connessione e comunicazione tra dispositivi e computer. È anche utilizzata per fornire alimentazione elettrica oltre che dati. È ampiamente utilizzata in vari dispositivi come smartphone, periferiche computer e dispositivi di archiviazione. È facile da usare, connessione plug-and-play.

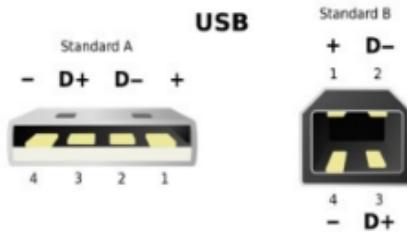


Figure 4: Porte USB

- Pin 1: Vcc, alimentazione 5V
- Pin 2: D-, Dati negativi
- Pin 3: D+, Dati positivi
- Pin 4: GND, Terra

I/O Mappato in Memoria

Ogni controller dispone di alcuni registri usati per le comunicazioni con la CPU. Scrivendo in questi registri, il sistema operativo può ordinare al dispositivo di inviare dati, accettarli, accendersi e spegnersi o eseguire qualche altra azione. Leggendo da questi registri, il sistema operativo apprende quale sia lo stato del dispositivo, se sia pronto ad accettare un nuovo comando e così via.

Oltre ai registri di controllo, molti dispositivi hanno un buffer di dati su cui il sistema operativo può scrivere e leggere. Ad esempio, un metodo classico dei computer per visualizzare i punti sullo schermo è avere una RAM video.

Port-Mapped I/O

Questo metodo assegna a ogni registro di controllo un numero di porta di I/O associato, di solito un intero di 8 o 16 bit. Per esempio, per leggere da una porta: `IN REG, PORT`, mentre per scrivere: `OUT PORT, REG`.

È importante avere una separazione degli spazi di indirizzi della memoria e dell’I/O. Quindi `IN R0, 4` legge dalla porta 4 e salva in R0, mentre `MOV R0, 4` legge dall’indirizzo di memoria 4 e salva in R0. I due “4” sono indirizzi diversi.

Memory-Mapped I/O

Questo metodo assegna a ogni registro di controllo un indirizzo di memoria univoco. I registri di controllo sono mappati nello spazio di memoria. Viene eliminata la necessità di istruzioni speciali di I/O come IN e OUT. Inoltre, i registri di controllo possono essere trattati come variabili in un

linguaggio come C, facilitando la scrittura di driver direttamente in C e non in assembly. Non c'è bisogno di una protezione complicata; il sistema operativo semplicemente non deve mettere questi indirizzi nello spazio degli indirizzi virtuali di qualunque processo utente. Attraverso la gestione delle pagine di memoria, è possibile dare un controllo selettivo su dispositivi specifici. Consente l'esecuzione di driver di dispositivi in spazi separati, aumentando la sicurezza e riducendo le dimensioni del kernel. Uno svantaggio di questo metodo è la gestione inefficiente delle cache e dei registri mappati in memoria.

Un possibile approccio ibrido è filtrare gli indirizzi per distinguere tra memoria e dispositivi di I/O. Gli indirizzi di I/O vengono inoltrati ai dispositivi anziché alla memoria.

Direct Memory Access (DMA)

Il DMA permette alla CPU di scambiare dati con i controller dei dispositivi bypassando il trasferimento manuale byte per byte, riducendo lo spreco di tempo alla CPU e migliorando l'efficienza del trasferimento dei dati.

Confrontiamo ora il trasferimento di dati tradizionale, ovvero senza DMA, e il trasferimento di dati con DMA.

- **Senza DMA:** Il controller del disco legge i dati e li memorizza nel suo buffer. Dopo aver controllato gli errori, provoca un interrupt e il sistema operativo copia i dati in memoria, sprecando tempo alla CPU.
- **Con DMA:**
 - Passo 1: La CPU impone al controller DMA e invia un comando al controller del disco.
 - Passo 2: Il controller DMA richiede la lettura al controller del disco.
 - Passo 3: Scrittura in memoria da parte del controller del disco.
 - Passo 4: Conferma dal controller del disco al DMA.

I passi 2-4 vengono ripetuti fino al completamento del trasferimento, al termine il DMA invia un interrupt alla CPU. In questo modo, la CPU delega il trasferimento dei dati al DMA, permettendo alla CPU di eseguire altre operazioni.

Modalità DMA e interazioni con il BUS:

- **Cycle Stealing:** DMA trasferisce una parola per volta, "rubando" cicli alla CPU nel caso in cui anche alla CPU servisse il BUS.
- **Modalità Burst:** DMA ottiene il controllo completo del bus, eseguendo trasferimenti multipli in una volta.
- **Fly-by Mode:** DMA trasferisce dati direttamente alla memoria principale senza intermediari.

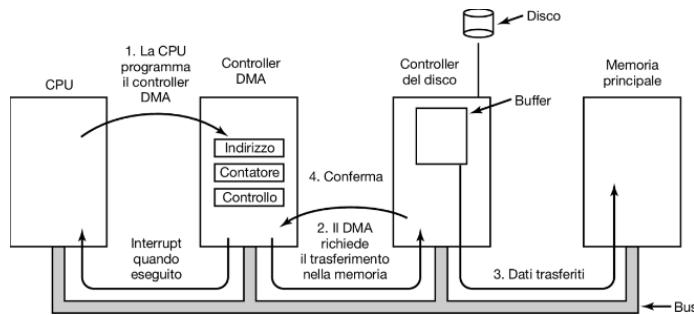


Figure 5: Funzionamento DMA

Interrupt

Gli interrupt possono essere causati da diverse cause:

- **Trap:** Azione deliberata da parte del codice del programma, ad esempio una trap nel kernel per una chiamata di sistema.
- **Fault o Eccezione:** Azioni non deliberate, come errori di segmentazione o divisione per 0.
- **Interrupt Hardware:** Segnali inviati da dispositivi come stampanti o reti della CPU.

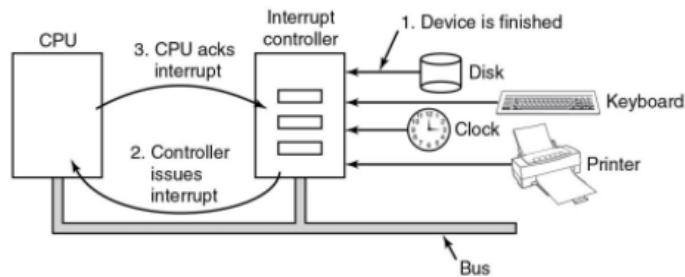


Figure 6: Cause dell'Interrupt

Un dispositivo di I/O invia un segnale di interrupt alla CPU mediante un bus, poi viene gestito dal chip nel controller degli interrupt sulla scheda madre. Se non ci sono altri interrupt in corso, il controller gestisce immediatamente l'interrupt; in caso di interrupt simultanei, il dispositivo è temporaneamente ignorato.

Processo di Gestione degli Interrupt

- **Segnalazione dell'Interrupt alla CPU:** Il controller assegna un numero alle linee degli indirizzi per specificare il dispositivo che richiede attenzione e invia un segnale di interruzione alla CPU.
- **Interruzione e Gestione da Parte della CPU:** La CPU interrompe il suo attuale compito. Utilizza un numero sulle linee degli indirizzi come indice nella tabella del vettore degli interrupt per ottenere un nuovo contatore di programma.
- **Il Vettore degli Interrupt:** Punta all'inizio della procedura di servizio degli interrupt corrispondente.
- **Conferma e Gestione degli Interrupt:** La procedura di servizio conferma l'interrupt scrivendo su una porta del controller degli interrupt.
- **Salvataggio dello Stato:** Al minimo, il contatore di programma deve essere salvato per riavviare i processi interrotti. Il salvataggio avviene nei registri interni o sullo stack. La CPU deve decidere se usare lo stack corrente del processo rischiando errori di pagina e puntatori illeciti o lo stack del kernel, producendo un overhead, poiché deve cambiare il contesto della MMU e possibile invalidazione della cache o del TLB.

Tipologie di Interrupt

1. **Interrupt Precisi:** Situazione in cui il sistema può determinare con esattezza quali istruzioni sono state completate al momento dell'interrupt e quali no. In questo caso, il PC è salvato in un luogo noto. Tutte le istruzioni eseguite prima del PC sono completate. Nessuna istruzione dopo il PC è stata eseguita. Lo stato dell'istruzione puntata dal PC è noto. Per gestire questo interrupt, la CPU cancella gli effetti di eventuali istruzioni transitorie eseguite dopo il PC, garantendo compatibilità e prevedibilità.
2. **Interrupt Imprecisi:** Condizione in cui diverse istruzioni vicino al contatore di programma si trovano in vari stati di completamento al momento dell'interrupt, rendendo incerto lo stato esatto del programma. Richiede che la CPU salvi una grande quantità di stato interno sullo stack, rendendo tutto il processo di gestione molto lento.

Sistemi Operativi

Ionut Zbirciog

19 December 2023

Principi del Software di I/O

Obiettivi

- **Indipendenza dal Dispositivo:** Il software di I/O dovrebbe permettere l'accesso a diversi dispositivi senza specificare il tipo di dispositivo in anticipo.
- **Denominazione Uniforme:** I nomi di file o dispositivi dovrebbero essere stringhe o numeri indipendenti dal dispositivo.
- **Gestione degli Errori:** Gli errori vanno gestiti il più vicino possibile all'hardware, idealmente dal controller stesso o dal driver del dispositivo. Errori transitori, come lettura da disco, spesso scompaiono ripetendo l'operazione.
- **Trasferimenti Sincroni vs Asincroni:** La maggior parte dell'I/O fisico è asincrono, ma per semplicità, molti programmi utente trattano l'I/O come se fosse sincrono. Il sistema operativo rende operazioni asincrone sembranti bloccanti, ma fornisce l'accesso all'I/O asincrono per applicazioni di alte prestazioni.
- **Buffering:** Spesso i dati da un dispositivo non vanno direttamente alla destinazione finale, richiedendo un buffer temporaneo. L'uso di buffer può influenzare le prestazioni, soprattutto per dispositivi con vincoli real-time.
- **Dispositivi Condivisibili vs Dedicati:** Dispositivi come dischi e SSD possono essere condivisi da più utenti, mentre altri come stampanti e scanner sono tipicamente dedicati.

Tipologie di Software per I/O

I/O Programmato

La CPU gestisce direttamente tutto il processo di trasferimento di dati. Un esempio pratico:

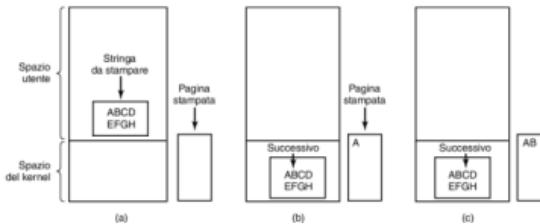


Figure 1: Processo di stampa con I/O programmato

Un processo utente prepara un stringa in un buffer dello spazio utente. Il processo effettua una chiamata di sistema per stampare la stringa, dopo aver ottenuto l'acceso alla stampante (a). Il sistema operativo copia il buffer in uno spazio kernel. Invia i caratteri ala stampante uno alla volta, aspettando che questa sia pronta per ogni carattere (b) e (c). Il sistema operativo entra in un ciclo di polling (busy waiting), controllando il registro di stato della stampante e inviando un carattere alla volta.

Occupava la CPU a tempo pieno durante il processo di I/O, facendo continuamente polling sullo stato della stampante. L'I/O programmato è efficace quando il tempo di elaborazione di un carattere è breve.

Il codice di esempio:

```
copy_from_user(buffer, p, count);
for(int i = 0; i < count; i++){
    while (*printer_status_reg != READY);
    *printer_data_register = p[i];
}
return_to_user();
```

I/O Guidato dagli Interrupt

Andiamo a considerare una stampante che non ha buffer, ma stampa un carateere dopo l'altro appena arriva, con un ritardo di 10 ms. In questo modo la CPU ha un tempo di 10 ms in cui potrebbe fare altro. Per permettere ciò, Dopo la chiamata di sistema per stampare la stringa, il buffer viene copiato nello spazio del kernel, come abbiamo mostrato prima, e il primo carattere è copiato nella stampante appena è in grado di accettarlo. A questo punto la CPU chiama lo scheduler e viene eseguito un altro processo. Il processo che ha richiesto la stampa della stringa è bloccato finché non è stampata l'intera stringa Quando la stampante ha stampato il carattere ed è pronta ad accettare il successivo, genera un interrupt che ferma il processo attuale e ne salva lo stato. Poi è eseguita la procedura di servizio di interrupt della stampante;

I/O con DMA

Uno svantaggio ovvio dell'I/O guidato dagli interrupt è che avviene un interrupt a ogni carattere. Gli interrupt richiedono tempo, pertanto questo schema spreca una certa quantità di tempo della CPU. Una soluzione è l'uso del DMA. In questo caso l'idea è di lasciare che il controller DMA invii i caratteri alla stampante uno alla volta, senza disturbare la CPU. In sostanza, il DMA è I/O programmato, solo che il lavoro è delegato al controller DMA anziché alla CPU principale. Questa strategia richiede un hardware speciale (il controller DMA), ma lascia che la CPU faccia altre cose durante l'I/O. Il grosso vantaggio del DMA consiste nella riduzione del numero degli interrupt da uno per carattere a uno per buffer stampato.

Struttura del Software di I/O

Il software di I/O è generalmente organizzato in quattro livelli, come mostrato nella figura, ciascuno dei quali ha una funzione ben definita da eseguire e un'interfaccia ben definita verso i livelli adiacenti.



Figure 2: Struttura del software di I/O

Gestore degli Interrupt

Passaggi nel software dopo il completamento dell'interrupt hardware:

1. Salvataggio di tutti i registri (incluso il PSW) non ancora salvati dall'interrupt hardware.
2. Impostazione di un contesto per la procedura di servizio dell'interrupt. Potrebbe implicare l'impostazione di TLB, MMU e una tabella delle pagine.
3. Impostazione di uno stack per la procedura di servizio dell'interrupt.
4. Conferma al controller degli interrupt. Qualora manchi il controller centralizzato degli interrupt, riabilitazione degli interrupt.
5. Copia dei registri da dove erano stati salvati (magari su qualche stack) alla tabella dei processi.
6. Esecuzione della procedura di servizio dell'interrupt. Tipicamente estrarrà informazioni dai registri del controller del dispositivo che ha generato l'interrupt.
7. Scelta di quale processo eseguire come successivo. Se l'interrupt ha fatto sì che qualche processo a priorità alta che era bloccato sia ora pronto, potrebbe essere scelto adesso per l'esecuzione.

8. Impostazione del contesto della MMU per il processo successivo da eseguire. Potrebbe essere anche necessario impostare il TLB.
9. Caricamento dei nuovi registri del processo, incluso il suo PSW.
10. Avvio dell'esecuzione del nuovo processo.

Driver dei Dispositivi

Il ruolo del driver è di gestire i dispositivi di I/O attraverso registri di dispositivi specifici. I driver oltre a gestire dispositivi specifici, possono anche gestire classi di dispositivi. Ogni dispositivo necessita di un codice specifico, noto come driver del dispositivo solitamente fornito dal produttore. Tecnologie come USC utilizzano una pila di driver per gestire una vasta gamma di dispositivi. I driver di solito fanno parte del kernel del sistema operativo per poter accedere ai registri del controller del dispositivo. Se sono usati nello spazio utente, sono più facili da installare, mettono meno a rischio il SO ma sono più lenti. Il sistema operativo deve permettere l'installazione di codice scritto da terze parti. I driver si posizionano sotto il resto del sistema operativo. In alcuni sistemi operativi, i driver sono inclusi nel programma binario del sistema operativo, nei sistemi moderni, i driver vengono caricati dinamicamente. I sistemi operativi classificano i driver in due categorie. A blocchi, come i dischi, a caratteri come le stampanti e le tastiere.

Il driver gestisce le scritture e le letture, inizializza il dispositivo, verifica la validità dei parametri di input, traduce i parametri in comandi specifici per il dispositivo.

Software del SO Indipendente dal Dispositivo

Il software di I/O indipendente dal dispositivo funge da intermediario tra i driver specifici dei dispositivi e le applicazioni utente. Mira a semplificare l'intereazione con i dispositivi hardware offrendo un interfaccia uniforme e gestendo operazioni comuni.

- Interfaccia Uniforme dei Driver dei Dispositivi: Fornisce un'interfaccia standard per diversi tipo di driver di dispositivo. Evita la necessità di modificare il sistema operativo ogni volta che viene introdotto un nuovo dispositivo. Standardizza un modello uniforme dove tutti i driver condividono la stessa interfaccia verso il SO. Ogni classe di dispositivi ha un insieme definito di funzioni che i driver devono superare.
- Buffering: Gestisce i buffer per l'efficienza del trasferimento dei dati tra i dispositivi e il sistema.
- Gestione degli Errori: Identifica e comunica gli errori provenienti dai dispositivi all'utente o ad altri sistemi.
- Gestione dei Dispositivi Dedicati: Gestisce l'assegnazione e la liberazione di dispositivi dedicati a specifici compiti o utenti.
- Uniformità della Dimensione dei Blocchi: Assicura che la dimensione dei blocchi di dati sia gestita in modo uniforme, indipendentemente dalla specifica del dispositivo.

Software per I/O a Livello Utente

Il ruolo delle librerie di I/O è di facilitare le chiamate di sistema per l'I/O, per esempio la funzione write() o read() in C. Funzioni come printf() e scanf() trasformano e gestiscono i dati prima di invocare funzioni di sistema, facilitando operazioni di input e output. Queste librerie semplificano la programmazione di I/O, permettendo ai programmatore di concentrarsi sulla logica dell'applicazione. Lo spooling è un modo per interagire con i dispositivi dedicati in un sistema multiprogrammato, evitando il blocco prolungato da parte di un processo. E' implementato utilizzando un processo daemon e una directory di spooling, per ordinare e gestire i lavori di stampa. Lo spooling incrementa l'efficienza nell'uso dei dispositivi dedicati e migliora la gestione delle risorse, permettendo a più processi di accedere ai dispositivi in modo sequenziale. Spooling: mettere in coda per la stampa.