Sistemi operativi

Un riassunto di
Paolo Alfano

Note Legali

Liberatoria, aggiornamenti, segnalazione errori:

Quest?opera viene pubblicata in formato elettronico senza alcuna garanzia di correttezza del suo contenuto. Il documento, nella sua interezza, è opera di Paolo Didier Alfano
e viene mantenuto e aggiornato dallo stesso, a cui possono essere inviate eventuali segnalazioni all'indirizzo *paul15193@hotmail.it*
Ultimo aggiornamento: 1 febbraio 2016

Contents

1 Introduzione

1.1 Cos'è il sistema operativo

Il sistema operativo (abbreviato SO, oppure OS in inglese) è lo strato software che gestisce le risorse di un computer per i suoi utenti e le applicazioni. Essendo di fondamentale importanza ci chiediamo quali funzioni debba avere un OS all'interno del computer:

1. Arbitro: poiché il sistema operativo deve gestire le risorse condivise, deve poter interrompere o avviare programmi, e isolare le applicazioni le une dalle altre. Isolare i programmi è fondamentale per aumentare la sicurezza del sistema e facilitare eventuali fasi di debugging.

2. Illusionista: i moderni sistemi operativi danno l'impressione di avere a disposizione infinità memoria o che quanto meno il processore sia del tutto dedicato alle attività gestite dall'utente. Ovviamente la situazione è leggermente più complicata.

3. Collante: il SO deve anche fornire una serie di servizi standard alle applicazioni per fare in modo che dati presi in una certo programma siano utilizzabili su altri programmi all'interno del computer.

Dunque quali dovrebbero essere i parametri da scegliere per valutare la bontà di un sistema operativo?

- Affidabilità e disponibilità: un SO è affidabile quando esegue esattamente le operazioni per cui è stato pensato. Un SO è tanto più disponibile quanto è maggiore la percentuale di volte in cui è utilizzabile.

- Sicurezza: il SO deve essere attaccabile il meno possibile da minacce esterne quali virus. Infatti per rendere sicuro un computer non basta avere un ottimo isolamento dei vari programmi.

- Portabilità: rappresenta la capacità di un SO di lavorare sul maggior numero possibile di configurazioni di hardware.

- Prestazioni: poiché le prestazioni di una macchina sono immediatamente visibili all'utente, è estramemente importante che la gestione dei vari programmi sia quanto più fluida e rapida possibile.

- Adottabilità: rappresenta la quantità di hardware e software che hanno deciso di adottare il SO come loro piattaforma di sviluppo.

2 L'astrazione del kernel

Per poter soddisfare alcune delle richieste viste nel paragrafo precedente, il computer è fornito di un *kernel*. Il kernel rappresenta il più basso livello software eseguibile su un sistema. Il principale vantaggio di avere un kernel consiste nel possedere una struttura capace di fornire protezione e di agire a livello del processore senza restrizioni.

2.1 L'astrazione del processo

La differenza tra programma e processo è che un programma si compone di diversi processi che possono essere in esecuzione. Allora chiamiamo *processo* l'istanza di un programma con diritti limitati. Il sistema operativo tiene traccia dei vari processi tramite una struttura detta PCB, Process Control Block. Il PCB (uno per ogni processo) memorizza tutte le informazioni necessarie al sistema operativo come la locazione del processo, i suoi privilegi, quale utente ne ha avviato l'esecuzione..

2.2 Operazioni Dual-Mode

Le istruzioni che il processore riceve possono provenire da programmi di ogni tipo, non per ultimi software che potrebbero risultare dannosi. Per avere una gestione sicura del sistema il processore può lavorare in due modalità distinte rappresentate da un bit nel registro di stato del processore:

- Modalità Kernel: è la modalità privilegiata in cui si ha l'accesso all'hardware senza restrizioni.
- Modalità Utente: consente di avere un accesso limitato all'hardware sotto stretto controllo del SO.

Affinché queste due modalità possano essere utilizzate il sistema operativo deve supportare

1. Istruzioni privilegiate: per evitare che le applicazioni a livello utente cambino i loro privilegi si impedisce ad ogni processo di cambiare l'insieme di bit che ne rappresentano il livello, ad esempio non consentendo la scrittura in celle di memoria non permesse. Dunque chiamiamo *istruzioni privilegiate* quelle istruzioni eseguibili in modalità kernel ma non in modalità utente.
2. Protezione della memoria: impedire ai programmi di scrivere su memoria non consentita tramite la protezione *base and bound* (in italiano base e limite), con cui ai processi è consentito operare in precise sezioni di memoria. Infatti ogni volta che il processore preleva un'istruzione controlla che il suo indirizzo sia compreso tra la base e il limite. Per diversi motivi, come la necessità di avere strutture dinamiche legate a rilocazioni del programma

in memoria, si preferisce utilizzare l'indirizzamento virtuale che imposta il base and bound a partire da zero per poi "traslarlo" di un numero di celle opportuno.

3. Interruzioni da timer: per evitare che un processo monopolizzi il processore, il sistema operativo deve possedere la capacità di riprenderne il controllo dopo un intervallo di tempo fissato.

2.3 Modi di trasferimento

Esistono sostanzialmente tre motivi per cui effettuare un cambio di contesto utente/kernel, solitamente indicato con il termine *trap*:

- Interruzione: l'interruzione è un segnale asincrono proveniente da un componente esterno al processore che potrebbe interessare il processore stesso.. Quando il processore riceve un'interruzione finisce di eseguire l'istruzione in corso e dopo aver salvato le impostazioni di un ipotetico processo interrotto a metà si dedica all'interruzione. E' da notare che sulle macchine multiprocessore l'interruzione si avvia su un solo processore mentre gli altri non si accorgono di niente. Ogni interruzione viene gestita in modo diverso dall'*interrupt handler*, un vettore di puntatori a diverse serie di istruzioni specifiche per ogni interruzione. L'interruzione viene anche usata per segnalare che un'attività su una periferica di input/output è terminata, ad esempio viene generata un'interruzione ogni volta che viene spostato o premuto un tasto del mouse.

- Eccezione: l'eccezione è un evento hardware causato da un comportamento dell'utente che causa un istantaneo cambio di contesto utente/kernel. Un tipico esempio di eccezione è la richiesta di un'applicazione di effettuare operazioni con privilegi che non gli appartengono.

- Chiamate di sistema: le chiamate di sistema sono le richieste volontarie dell'utente di accedere a funzioni permesse del kernel. In tal caso si ha un cambio di contesto dopo il quale il kernel effettua le operazioni richieste. Tipici esempi di system calls sono la creazione di una connessione ad un server web, la lettura, la scrittura e la cancellazione di file.

Allo stesso tempo è possibile che si richieda di effettuare un cambio di contesto kernel/utente. Questo può succedere se:

- dopo la creazione di un nuovo processo si vuole passare il controllo dell'applicazione all'utente.

- si rientra dalla gestione di un'interruzione, di un'eccezione o di una system call.

- se si vuole cambiare il processo in esecuzione sul processore, come nella creazione del nuovo processo, dopo si passa il controllo all'utente.

- viene notificata una *upcall* all'utente, cioè un avvertimento proveniente da un programma.

2.4 Implementare un cambio di contesto sicuro

Per gestire al meglio i cambi di contesto, in generale si utilizza un vettore delle interruzioni in cui sono memorizzate le istruzioni necessarie per gestire al meglio i tre possibili motivi che generano un cambio di contesto utente/kernel.
Inoltre si utilizza un'area di memoria del kernel detta *interrupt stack* in cui il processore salva le informazioni dei processi interrotti a causa del cambio di contesto prima di passare alla gestione dell'interruzione stessa.
In realtà molti sistemi operativi decidono di dedicare due stack ad ogni processo: una a livello utente e una a livello kernel per facilitare il cambio del controllo del processore tra processi diversi.
Infine, per evitare sovrapposizioni di interruzioni, eccezioni e system calls, durante la gestione di un'interruzione vengono temporaneamente ignorate tutte le altre. Questo procedimento, detto *interrupt masking* deve durare il meno possibile per poter operare nuovamente in caso di situazioni più gravi da gestire.

2.5 Modalità di trasferimento in x86

2.6 Implementare system calls sicure

Poiché le system calls possiedono uno o più argomenti, sono un facile mezzo per accedere in modo maligno alle funzionalità del kernel tramite argomenti corrotti. Per questo motivo il kernel tratta gli argomenti delle system calls come se fossero estremamente pericolosi inserendo due strutture intermedie dette *user stub* e *kernel stub* atte a controllare la bontà degli argomenti passati. In particolare il kernel stub ha quattro compiti

- stabilire che la locazione degli argomenti non sia corrotta.
- controllare che i parametri, come il nome del file, non si estandano in aree di memoria non concesse.
- copiare gli argomenti prima di controllarli. E' necessario ricontrollare che gli argomenti siano gli stessi (confrontandoli con quelli copiati all'inizio) per evitare attacchi del tipo *time of check vs. time of use* in cui gli argomenti vengono modificati dopo il controllo.
- restituire i risultati della system calls

2.7 Avviare un processo

Avviare un processo a livello utente è un'operazione che si compone di diverse fasi da noi già viste: allocare e iniziallizzare il PCB, allocare la memoria per il processo, copiare il programma dal disco nelle nuove locazioni di memoria, allocare una stack a livello utente e una a livello kernel, copiare gli argomenti

del processo in memoria(un argomento potrebbe essere il nome del file), e infine trasferire il controllo all'utente.

2.8 Implementare upcalls

Vi sono diversi motivi per i quali potrebbe essere necessario notificare una upcall. Alcuni di questi sono: la segnalazione del completamento di un'operazione di I/O, la comunicazione che un processo ha terminato un certo lavoro ad un altro processo, oppure la necessità di informare l'utente che si è verificata un'eccezione come una divisione per zero..
In realtà dal punto di vista dell'implementazione, le upcalls presentano numerose analogie con le interruzioni, basandosi su un sistema di *handlers* di *masking* e di salvataggi di processi interrotti in locazioni di memoria stabilite. La segnalazione vera e propria all'utente viene fatta resistuendo dei messaggi di avviso all'utente tramite l'handler adibito a quella specifica upcall.

2.9 Caricamento di un sistema operativo

La prima operazione che viene eseguita all'avvio del computer è il caricamento di alcune funzioni fondamentali contenute nel BIOS, scritto sulla ROM (read only memory). Ci si potrebbe chiedere perché non scrivere il kernel nella ROM. Una gestione di questo tipo è solitamente evitata perché ciò che viene scritto sulla ROM è solitamente immutabile(read only) e dunque sarebbe pressoché impossibile effettuare modifiche al kernel in caso di mafunzionamenti. Successivamente il BIOS copia il bootloader che a sua volta caricherà il kernel del sistema operativo dal quale poi è possibile caricare le applicazioni di login e tutto il resto. Ovviamente il bootloader è memorizzato con una *firma crittografica* che viene controllata dal BIOS per garantire maggiore sicurezza.

2.10 Macchine virtuali

Alcuni sistemi operativi forniscono un'ulteriore livello di astrazione attraverso intere macchine virtuali a livello utente. Il sistema che fornisce un servizio del genere è detto *sistema operativo host* mentre il sistema operativo della macchina virtuale è detto *sistema operativo guest*. Quando l'ospite viene avviato, esso procede come se fosse stato caricato dal bootloader. Le operazioni eseguite in una macchina virtuale che richiedono operazioni di tipo privilegiato vengono gestite dal sistema host che funge da mediatore tra hardware e sistema operativo ospite. Ad esempio quando in una macchina virtuale viene effettuata una system call, essa viene gestita dal sistema host.

3 L'interfaccia di programmazione

Ogni sistema operativo è dotato di diverse funzionalità atte a risolvere alcuni problemi tipici come la gestione dei processi dei threads e della memoria, l'input/output, l'archiviazione dei file..
Una domanda interessante da porsi è

"A quale livello dovrei implementare queste funzionalità?"

La risposta è solitamente un compromesso tra flessibilità, sicurezza, affidabilità e prestazioni. Vediamole più nel dettaglio

- Flessibilità: se dovessimo scegliere in base alla flessibilità del sistema sarebbe preferibile collocare più funzionalità possibili a livello utente per rendere più facili le operazioni di modifica del codice.
- Sicurezza: dal punto di vista della sicurezza è preferibile collocare le funzionalità a livello kernel perché se i controlli di protezione fossero collocati in una libreria a livello utente, potrebbero essere scavalcati dalle applicazioni.
- Affidabilità: secondo i criteri di affidabilità sarebbe meglio collocare le funzionalità a livello utente per avere un codice meno complesso.
- Prestazioni: infine, effettuare passaggi di contesto è un'operazione che ha un costo computazionale quindi per evitarne il più possibile sarebbe meglio implementare le funzionalità a livello kernel.

Nel complesso, non esiste una risposta definitiva alla domanda. La risposta deve essere cercata nelle caratteristiche che si richiedono maggiormente tra le quattro sopra elencate.

3.1 Gestione del processo

Le varie operazioni che vengono effettuate sui processi sono gestiti in modo diverso a seconda del sistema operativo. Windows ad esempio, per avviare un nuovo processo utilizza una funzione CreateProcess che possiede molti argomenti in ingresso. UNIX invece preferisce utilizzare un set di quattro funzioni:

1. Fork: è una funzione che serve ad effettuare una copia del *processo padre*. Per fare ciò viene allocato lo spazio necessario per il PCB e per il contenuto della copia. Il *processo figlio* è una copia esatta del padre ad eccezione per una chiave che serve per contraddistinguerli. Infatti la funzione fork restituisce al padre il PID (process identifier) del figlio, mentre al figlio restituisce 0 per rappresentare il successo. Se si verifica un errore durante l'esecuzione la fork restituisce un valore negativo. Le modifiche necessarie a creare il contesto vengono fatte successivamente. Inoltre la funzione fork informa lo scheduler che il nuovo processo è pronto ad avviarsi. In ogni caso non ci è dato sapere chi tra padre e figlio verrà eseguito prima

perché dipende dalla politica di scheduling. In generale la fork non riceve argomenti

2. Exec: solitamente chiamata dal figlio, serve a caricare l'immagine eseguibile (formata da programma e argomenti) in memoria, e a creare il contesto modificando il PCB per cominciare l'esecuzione. In generale la exec riceve due argomenti in ingresso, il nome del programma e l'array degli argomenti.

3. Wait: serve per interrompere il prcesso padre fino a quando il processo figlio non ha terminato. E' da notare che comunque, in UNIX la funzione wait non è chiamata obbligatoriamente.

4. Signal: serve per notificare tramite una upcall la fine di un processo, la temporanea sospensione a causa di un debug, la ripresa dopo un debug..

3.2 Input/Output

Per far fronte alla diversità strutturale delle periferiche, in UNIX è stato deciso di fornire un'interfaccia comune fornendo un set predefinito di system calls (open, close, read, write..). Ad esempio tutte le applicazioni che vogliono effettuare I/O devono chiamare la funzione open sul dispositivo da utilizzare. La funzione open restituisce un *file descriptor* riguardante la periferica che servirà per utilizzi successivi delle funzioni read, write, close..
Esistono anche altre funzioni standard come la *close* con la quale il processo esplicita di aver finito di utilizzare un certo dispositivo.
Più importante è la *UNIX pipe*, un kernel buffer, o memoria di transito del kernel, avenete due file descriptor, uno in lettura e uno in scrittura. Viene solitamente utilizzata da un produttore e da un consumatore e viene chiusa una volta terminato lo scambio.

3.3 Implementare una shell

3.4 Comunicazione tra processi

3.5 Struttura del SO

Abbiamo cominciato questo capitolo chiedendoci a quale livello implementare le varie funzionalità del sistema operativo. In generale esistono due divsioni sul tipo di kernel del sistema operativo:

- Kernel monolitico: con una scelta di questo tipo si preferisce inserire gran parte delle funzioni all'interno del kernel mettendo in evidenza due strutture del kernel monolitico:

    1. *Harware Abstraction layer*: detto anche HAL è uno strato software pensato per mettere in comunicazione il kernel con una gran varietà di macchine ed architetture. Con un HAL ben strutturato il SO risulta essere indipendente dal tipo di processore e di macchina.

2. *Driver dei dispositivi dinamici*: è un insieme di software che consente l'inserimento di nuovi driver per nuovi dispositivi. Poiché tramite i driver si potrebbero effettuare attacchi al sistema vengono sempre sottoposti a controlli e per migliorare l'esperienza dell'utente è possibile inviare dei dati al produttore del driver per comprendere meglio eventuali problemi occorsi. Comunque, sia Microsoft che Apple incoraggiano fortemente gli sviluppatori software a far funzionarei driver a livello utente di modo che, in caso di problemi, questi ultimi riguardino solo le strutture dati interne dei driver.

- Microkernel: l'approccio microkernel è radicalmente diverso rispetto a quello del kernel monolitico in quanto minimizzando le funzioni del sistema operativo si delega il lavoro all'utente che difficilmente riuscirebbe ad implementare le funzioni necessarie e che anche se ci riuscisse lo farebbe in modo meno efficiente.

4 Concorrenza e threads

Utilizzeremo la parola *concorrenza* per riferirci ad attività multiple che possono avvenire nello stesso momento. Gestire correttamente la concorrenza è una sfida chiave per gli sviluppatori di sistemi operativi. L'idea è infatti quella di scrivere un programma come un gruppo di flussi sequenziali, detti *threads*, che interagiscono e condividono risultati in modo preciso. Ovviamente una gestione di questo tipo richiede la scrittura di codice addizionale per coordinare i threads.

4.1 Casi d'uso dei threads

L'intuizione dietro all'utilizzo dei threads è semplice: in un programma possiamo rappresentare ogni compito come un thread. Possiamo infatti considerare il classico programma come un programma a thread singolo con una sola sequenza logica. Nella programmazione multi-thread invece ogni thread segue una singola sequenza di passi. Un esempio può essere quello di google earth in cui diverse sezioni della mappa sono rappresentate da thread, un altro thread visualizza i widget e infne un altro esegue le operazioni di caricamento. Usare i thread comporta essenzialmente quattro vantaggi:

- Esprimere logicamente compiti concorrenti: di fronte al dover programmare una grande quantità di compiti, risulta logicamente più comodo programmare separatamente ogni compito piuttosto che scrivere un'unica sequenza che intervalli l'escuzione dei vari compiti.
- Rapidità di risposta: dividendo i compiti è possibile spostarne alcuni in background di modo che l'utente non debba necessariamente attenderne il risultato. In generale i sistemi operativi fanno un uso massiccio dei thread per preservare le prestazioni. I sistemi operativi sono infatti programmati di modo che in media la rapidità di risposta sia elevata.
- Performance dei processori: ovviamente i programmi che utilizzano i threads su macchine multicore sono più efficienti, infatti possono fare lo stesso lavoro in meno tempo o una maggiore quantità di lavoro a parità di tempo.
- Performance sull'I/O: tramite i threads è possibile eseguire compiti mentre altri sono in attesa di un risultato da una periferica di I/O. Questa scelta è dettata da due fattori: in primo luogo le periferiche di I/O con cui i threads lavorano sono estremamente più lente e tenere in attesa altri processi sarebbe un notevole spreco di risorse. In secondo luogo l'arrivo di un dato/risultato da I/O è impredicibile dunque il sistema operativo deve essere capace di rispondere ad altre esigenze nel frattempo.

4.2 L'astrazione del thread

Prima di procedere diamo una definizione precisa di thread.
Un *thread* è la singola sequenza di esecuzione che rappresenta un compito processabile separatamente. Bisogna dire che ogni sistema operativo possiede un

thread scheduler che stabilisce quali threads sono in esecuzione e quali no. In generale però, non è possibile fare assunzioni sul tempo di esecuzione di una sequenza di threads in quanto le possibili esecuzioni di tale sequenza sono troppe a causa della possibile intromissione di eventi esterni come interruzioni o altro.

4.3 Simple thread API

Per utilizzare i threads sono solitamente fornite alcune funzioni:

- void thread_create(thread, func, arg): crea un nuovo thread e concorrentemente il thread chiamato esegue la funzione func passandogli gli argomenti arg.
- void thread_yield(): con questa funzione il thread cede volontariamente il processore.
- int thread_join(thread): aspetta che il thread restituisca il risultato e lo passa alla funzione exit
- void thread_exit(ret): termina il thread corrente salvando il valore ret nella struttura dati del thread corrente. Se un altro thread è in attesa di eseguire una join lo esegue.

E' da notare il parallelismo tra queste funzioni e le funzioni standard di UNIX già viste per creare i processi. Ad esmpio, tramite la funzione create un thread può creare un thread figlio che svolga un lavoro.

4.4 Struttura dati e ciclo vitale del thread

Proprio come succedeva per i processi esiste una struttura dati per rappresentare lo stato di un thread. Questa struttura è chiamata *thread control block*(TCB). Ogni TCB contiene:
una stack per contenere le informazioni necessarie per le procedure che il thread stesso utilizza. Notiamo anche che ogni thread può essere in esecuzione in un momento diverso in una locazione diversa con argomenti diversi, dunque ogni thread necessita del proprio stack. Ogni volta che il processore interrompe un thread in esecuzione ne salva lo stato attuale in una zona di memoria dedicata, solitamente la cima dello stack mentre il TCB del processo contiene il puntatore agli indirizzi suddetti.
Inoltre il TCB contiene altre informazioni utili per gestire il thread, per esempio il suo ID il suo stato o la sua priorità.
Al contrario di quanto detto in questo paragrafo in cui ogni processo possiede il proprio stato, esistono alcuni stati, detti *condivisi* che sono comuni a più thread di uno stesso processo oppure in comune con il sistema operativo.

4.5 Ciclo di vita del thread

Può essere utile considerare la progressione di un thread attraverso gli stati che lo possono caratterizzare: Essi sono sostanzialmente cinque:
init: la creazione di un thread lo pone nello stato di init allocandogli e inizializzandogli le strutture dati. Fatto questo, il suo stato passa a pronto e viene aggiunto alla *ready list*.
ready: un thread si trova in stato di pronto quando è pronto ad essere eseguito. In un qualunque momento potrebbe passare in esecuzione.
running: un thread è in questo stato quando è in esecuzione sul processore. Da questo stato, un thread può tornare in stato di pronto se esegue una yield o se viene costretto a rilasciare il processore dal kernel.
waiting: un thread in questo stato è in attesa che occorra un evento. Viene allora inserito nella *waiting list* consentendo ad un altro thread di andare in esecuzione.
finished: un thread che si trova in questo stato non viene più eseguito. Può essere temporaneamente collocato sulla *finished list* per altri scopi.

4.6 Implementare Kernel Threads

Il modo più semplice per implementare i threads è quello di includerli nel sistema operativo. Ovviamente un sistema che supporta i kernel threads supporta l'utilizzo di processi a thread singolo e processi a thread multipli. Inoltre per consentire le chiamate di sistema, si utilizzano dei threads a livello utente tramite le funzioni del paragrafo 4.3
In generale per creare un thread si eseguono alcune operazioni:

- Si alloca lo spazio per il TCB e lo stack del thread.
- Si iniziallizza il TCB impostando i parametri necessari affinché, quando il processo va in esecuzione, esegua la funzione func(arg) con gli argomenti arg. In realtà viene chiamata una stub intermedia che si occupa di restituire eventuali risultati di func e di chiamare la thread_exit.
- Si setta il thread su ready e lo si pone sulla ready list

Per cancellare un thread, quando viene chiamata un thread_exit si eseguono invece due passi: lo si rimuove dalla ready list di modo che non venga mai più eseguito, e si dealloca TCB e stack. Bisogna comunque fare attenzione perché se un thread cerca di rimuoversi e nel frattempo occorre un'interruzione successivamente potrebbe non essere più in grado di farlo. Allora si preferisce fare in modo che un thread non cancelli mai il proprio stato, ma solo lo stato altrui.
Per implementare i threads è anche necessario strutturare il cambio di contesto per i threads. Il cambio di contesto si può verificare per due motivi:

- Cambio di contesto volontario: si verifica quando il thread chiama la funzione thread_yield. In particolare la yield disabilita le interruzioni e fa ripartire la ready list, se vi è qualcuno.

- Cambio di contesto non volontario: è il caso in cui si riceva un'interruzione o altro. In tal caso si salva lo stato del thread che era in esecuzione, si esegue un handler appropriato e successivamente si ripristina il successivo thread della ready list.

4.7 Combinare Kernel Threads e processi utente a thread singolo

Quando vengono utilizzati i processi, può accadere che essi siano a thread singolo o a thread multipli.
Fintanto che il processo contiene più di un thread il PCB contiene più informazioni di un TCB: ad esempio informazioni riguardo gli indirizzi, i cambi di contesto che sono avvenuti e altro. Se invece il processo è a thread singolo, allora la ready list può contenerlo. Ovvero la ready list contiene un misto di threads, provenienti da processi a threads multipli, e di processi a thread singolo, senza distinzione. Infatti una volta creato, il processo a thread singolo si comporta esattamente come un thread.

4.8 Implementare processi multithread

Uno dei modi per implementare i threads è quello che abbiamo visto utilizzando i kernel threads e le funzioni di supporto.
Un'alternativa è quella di implementare completamente i threads a livello utente(sono solitamente chiamati *green threads*). In realtà le prime implementazioni dei threads vennero fatte proprio in questo modo in quanto all'inizio ben pochi sistemi operativi supportavano l'utilizzo dei threads. L'idea è molto semplice: la libreria dei threads istanzia tutto ciò che è necessario ai threads, TCB, ready list, finshed list. Le waiting list come dati interni ai processi. Dopo, le chiamate alla libreria sono soltanto chiamate di procedure. Per il sistema operativo tutti i processi sembreranno normali processi a thread singolo. L'unica differenza è che vedendo un insieme di threads come un unico thread, quando uno dei threads viene sospeso, viene sospeso l'intero processo.
E' possibile anche implementare il prerilascio in modo molto simile a quello visto a livello kernel.
In realtà oggi spesso si utilizza un modello ibrido per combinare i vantaggi sia dello sviluppo a livello utente che a livello kernel, ad esempio evitando le transizioni a livello kernel. E' anche possibile adattare i thread implementati a livello utente sulle architetture multiprocessore facendo in modo che la libreria crei un kernel thread per ogni processore presente. In questo caso gli svantaggi dei green threads restano, dunque si è deciso di implementare dei meccanismi specifici per processare threads a livello utente. In Windows ad esempio esiste lo *scheduler activations* che viene attivato ogni volta che un evento kernel potrebbe riguardare le attività del livello utente, ad esempio se un thread si blocca in una system call viene notificato al livello utente che si dovrebbe scegliere un altro thread da eseguire. Il meccanismo è simile a quello già visto basato su liste ready,

waiting e altre, la differenza sta nel fatto che i risultati non sono restituiti al kernel ma al livello utente.

4.9 Astrazioni alternative

5 Accesso sincronizzato ad oggetti condivisi

Per implementare in modo efficiente la concorrenza tra threads cominciamo con l'assumere che esistano threads cooperanti che leggono e scrivono su stati di memoria condivisi. Sfortunatamente scrivere programmi con threads cooperanti risulta essere più difficile perché molti programmatori sono abituati a ragionare in modo sequenziale. Il modello di ragionamento sequenziale non funziona nel caso di threads cooperanti per diversi motivi: intanto perché l'esecuzione di un programma dipende dall'ordine di accesso alle varibili condivise, inoltre l'esecuzione di un programma è generalmente non deterministica a causa di molti fattori quali politica di scheduling, diversa frequenza di clock del processore..
Infine può capitare che il compilatore decida di riordinare le istruzioni in modo automatico per rendere più efficiente il programma.
Dunque per implementare in modo efficiente la concorrenza tra threads si introducono delle *strutture di sincronizzazione* che vedremo nei prossimi paragrafi.

5.1 Sfide

Il primo concetto che introduciamo per fare programmazione multi-threads è quello di *condizione di gara* che si verifica quando il comportamento di un programma dipende dalla sequenza di esecuzione dei threads. Ovviamente l'esecuzione di un processo, ovvero il risultato di una serie di operazioni effettuate tramite i threads, dipende da chi "vince" la gara. Il secondo concetto è quello delle *operazioni atomiche*, operazioni indivisibili che non possono essere intervallate ad altre.

5.2 Strutturare oggetti condivisi

Solitamente in un processo multi-threads, la cooperazione tra questi ultimi viene effettuata per mezzo di *oggetti condivisi*. Gli oggetti condivisi sono oggetti a cui possono accedere più threads cooperanti.
Implementare oggetti condivisi è un'operazione che consiste nell'avere variabili condivise e una struttura che sincronizzi l'accesso a tali variabili.
Infatti definiamo le *variabili di sincronizzazione* come delle strutture dati che coordinano gli accessi a stati condivisi. Le tre più note variabili di sincronizzazione sono:

1. Locks
2. Variabili di condizione
3. Semafori

che verranno viste nei prossimi paragrafi.

5.3 Locks: esclusione mutua

Una *lock* è una variabile di sincronizzazione che fornisce la mutua esclusione. Con questo si intende dire che quando un thread possiede la lock nessun altro thread può averla. La lock può essere posta a "guardia" di stati condivisi per fare in modo che solo un thread alla volta possa modificarli.
La lock abilita il concetto di mutua esclusione tramite due funzioni distinte: Lock::acquire() e Lock::release(). Questi metodi sono definiti come segue:

- Una lock può essere in uno solo di due stati: impegnato o libero.
- La lock è inizialmente in stato libero.
- Lock::acquire() attende che la lock sia in stato libero per poi acquisirla automaticamente. Sottolineiamo che l'operazione di controllo dello stato della lock e la sua acquisizione sono operazioni atomiche e che quindi un solo thread si accorge che la lock è utilizzabile e la acquisisce. Quale thread acquisirà la lock dipende dalla politica di scheduling.
- Lock::release setta la lock su stato libero.

In modo più formale la funzione lock può essere definita tramite tre proprietà:

1. Esclusione mutua: un solo thread alla volta può possedere la lock.
2. Progresso: se nessun thread possiede la lock e almeno un thread tenta di acquisirla, allora un thread avrà successo nell'ottenerla.
3. Attesa limitata: se un thread T prova ad acquisire la lock allora esiste un limite di volte in cui altri threads acquisiscono la lock prima di T.

Strettamente legato al concetto di lock abbiamo quello di *sezione critica*. Una sezione critica è una porzione di codice che accede in modo atomico a stati condivisi. Solitamente ad ogni sezione critica viene associata una lock per fare in modo che solo un thread per volta vi acceda.

5.4 Variabili di condizione: aspettando un cambiamento

Le *variabili di condizione* forniscono a un thread il modo di aspettare che un altro thread svolga una qualche azione. Alcuni esempi semplici sono l'attesa di un server di nuove richieste, un word processor che attende l'inserimento di un nuovo carattere..
In ognuno di questi esempi vogliamo che un thread attenda che una qualche azione cambi lo stato del sistema per avere un progresso. Un metodo per fare ciò potrebbe consistere nel controllare periodicamente se lo stato è cambiato. Sfortunatamente questa è una soluzione inefficiente in quanto il thread cicla continuamente consumando cicli del processore senza ottenere alcun progresso. Le variabili di condizione sono delle variabili di sincronizzazione che funzionano in tre modi:

- CV::wait(Lock *lock): la chiamata della funzione wait viene effettuata da un thread che possiede la lock ma si accorge di dover attendere qualcosa. A quel punto rilascia in modo atomico la lock per essere inserito in una lista di attesa delle varibili di condizione.

- CV::signal(): questa funzione prende un thread in lista di attesa e lo segna come abilitato ad essere eseguito. Se nessun thread è in lista di attesa, la signal non fa nulla.

- CV::broadcast(): la funzione broadcast prende tutti gli elementi sulla lista di attesa e li segna come abilitati ad essere eseguiti. Chi verà poi eseguito dipende da caso a caso. Se nessun thread è in lista di attesa, la broadcast non fa nulla.

Ovviamente si decide di richiamare la signal o la broadcast perché si suppone che qualcosa nell'interesse di un certo thread sia cambiato. Dobbiamo adesso fare alcune considerazioni per le variabili di condizione:
Le variabili di condizione sono *memoryless*, cioè ad eccezione per la lista di attesa non hanno uno stato interno per ricordare chi o quante volte ha chiamato le funzioni wait, signal e broadcast. Sottolineiamo inoltre che la funzione wait rilascia in modo atomico la lock. Infatti un thread che ceda la lock prima di chiamare la wait(dunque che cede la lock senza essere messo in lista) potrebbe rimanere in attesa di riottenere la lock per sempre. Infine dobbiamo considerare che quando un thread in attesa viene abilitato all'esecuzione dalla signal o dalla broadcast potrebbe non essere eseguito subito in quanto il thread viene semplicemente messo in una lista di thread pronti all'esecuzione e che quindi possa passare del tempo prima che ottenga la lock. Dunque può anche succedere che certe condizioni siano verificate quando viene chiamata una signal, ma che tali condizioni non sussitano più quando il thread viene eseguito, generando problemi.
Notiamo che a questo punto è necessaria una leggera rivisitazione del ciclo di vita dei threads visto nel capitolo precedente. L'utilizzo della lock e delle altre funzioni inserisce i threads negli stati di attesa, pronto e in esecuzione.

5.5 Progettare e implementare oggetti condivisi

La prima domanda da porsi quando si decide di scrivere un programma che supporti gli oggetti condivisi è: da dove cominciare?
La base di partenza è quella dei programmi single thread, ovvero si parte dalla decomposizione del problema in più oggetti di cui definisco un'interfaccia, ne identifico il giusto stato interno e le procedure che lo modifichino in modo appropriato.
Nella programmazione multithread si tratta di aggiungere alcuni passi:

1. Aggiungere una lock: ogni oggetto del problema deve avere una propria lock per consentire l'accesso esclusivo agli stati condivisi dell'oggetto.

2. Aggiungere il codice per acquire e release: il metodo più comune ed intuitivo è quello di acquisire la lock all'inizio di ogni procedura e rilasciarla alla fine di ognuna.

3. Identificare ed aggiungere le variabili di condizione: l'approccio più comune è quello di aggiungere una variabile di condizione ogni volta che il processo potrebbe ritrovarsi in attesa.

4. Aggiungere un loop per ogni wait: non bisogna dimenticarsi che ogni chiamata della wait deve essere inserita in un ciclo while che periodicamente verifichi la variabile di condizione.

5. Aggiungere le chiamate di signal o broadcast: per inserire correttamente tali chiamate basta chidersi "la chiamata di questa pocedura consente ad un altro thread di procedere?" ed inserire se necessario signal o broadcast.

In generale, gli autori del libro consigliano di seguire alcune semplici regole per implementare al meglio gli oggetti condivisi, e sono:

1. Avere una struttura logica coerente: si consiglia di utilizzare sempre una struttura che sia logicamente coerente e possibilmente standard.

2. Sincronizza utilizzando sempre le locks e le variabili di condizione: non perché le varibili di condizione consentano di rappresentare una maggior quantità di problemi ma perché per uno stesso problema, l'utilizzo di variabili di condizione e di lock rende il codice più chiaro rispetto all'utilizzo dei semafori.

3. Acquisire sempre la lock all'inizio di una procedura e rilasciarla alla fine: estendiamo il concetto di struttura logica standard del punto 1 sostenendo che operare in modo differente, ad esempio ponendo una lock release in mezzo a una procedura, rende più intricato il codice e più complesso il debug.

4. Tenere sempre la lock mentre si opera su variabili di condizione: infatti le variabili di condizione sono del tutto inutili senza stati condivisi che a loro volta poggino sul sistema delle lock.

5. Utilizzare wait in un loop: nella logica standard si prevede l'utilizzo della funzione wait all'interno di un while perché consente di avere un'incredibile libertà rispetto a dove inserire una signal in altre procedure.

6. Non usare praticamente mai la funzione thread_sleep: per porre un thread in attesa il metodo standard corretto è quello di utilizzare le wait se non in casi molto particolari.

5.6 Tre casi di studio

5.7 Implementare la sincronizzazione tra oggetti

Per implementare efficacemente la sincronizzazione tra oggetti facciamo una distinzione tra lock uniprocessore e lock multiprocessore. Nel caso di una macchina uniprocessore la scelta più ovvia potrebbe sembrare quella di disabilitare le interruzioni e riattivarle una volta che viene rilasciata la lock. In realtà una scelta di questo tipo potrebbe risultare pericolosa perché la porzione di codice a cui è associata una certa lock risulta essere particolarmente lungo, la macchina potrebbe non essere pronta a rispondere ad altre urgenze. Per un approccio migliore ricordiamo che una lock può esistere solo in due stati, libera o occupata. Si decide allora di disabilitare le interruzioni per un breve periodo di tempo durante il quale la lock viene acquisita. Ogni altro thread che cerchi di acquisire la lock viene posto in una coda ove rimarrà fino a quando un altro thread chiamerà una lock::release.
Nel caso in cui la macchina sia multiprocessore si utilizza una *spinlock*. Sarebbe a dire che tramite un'istruzione particolare detta *test and set* si controlla periodicamente -ciclando attivamente- se la lock si sia liberata. Questo approcio è inefficente ma d'altronde è impossibile eliminare completamente l'attesa attiva su una macchina multiprocessore.

5.8 Semafori considerati dannosi

Tra le possibili variabili di sincronizzazione che si sono succedute nel tempo hanno ricevuto particolare successo i *semafori*.
Un semaforo possiede:

1. un valore non negativo
2. una coda a priorità FIFO

I semafori sono poi definiti come segue:

- quando il semaforo viene creato il suo valore può essere iniziallizzato a un qualunque valore non negativo
- la funzione Semaphore::P() decrementa il valore associato al semaforo. Se tale decremento renderebbe negativo il valore del semaforo, il thread chiamante viene posto in coda al semaforo
- la funzione Semaphore::V() incrementa il valore del semaforo. Se vi sono thread in coda al semaforo il primo di questi(politica FIFO) viene posto nello stato ready

Tramite i semafori è possibile implementare la mutua esclusione. Basta porre il valore del semaforo uguale ad 1. Da questo momento in poi la Semaphore::P() è equivalente ad una Lock::acquire mentre la Semaphore::V() è equivalente ad una Lock::release

6 Sincronizzazione multi-oggetto

6.1 Performance di una lock multiprocessore

6.2 Patterns per il design di una lock

6.3 Contesa fra lock

6.4 Atomicità multi-oggetto

6.5 Deadlock

Una sfida che si incontra nel costruire programmi multi-thread è quella di evitare le *deadlock*. Una deadlock -ai fini pratici- è una situazione di stallo in cui un gruppo di thread non può più progredire. Più formalmente una deadlock è un ciclo di waiting fra thread dove ognuno di essi è in attesa che qualcun altro faccia qualcosa. L'esempio più semplice di deadlock è quello in cui un primo thread cerca di acquisire due diverse lock e un altro thread cerca di acquisire le due stesse lock in ordine inverso. Potrebbe succedere che entrambi i thread acquisiscano una delle due lock ma che non possano acquisire l'altra, dando luogo a una deadlock.
La differenza tra la condizione di deadlock e la condizione di *starvation* risiede nel fatto che nella starvation un singolo thread non progredisce per un periodo indefinito di tempo mentre nella deadlock ho un intero gruppo di thread che forma un ciclo nel quale nessuno progredisce poiché in attesa di qualcun altro. Dunque, la presenza di una deadlock implica la presenza di una starvation, ma non è vero il viceversa.
In generale, vi sono quattro diverse condizioni che potrebbero dar luogo ad una deadlock:

- risorse limitate: se ho un numero finito di risorse e diversi thread che vi vogliono accedere.
- non avere prerilascio: quando un thread acquisisce una risorsa se non gli può essere tolta fino a che non la rilascia.
- attesa mantenendo una lock: se un thread si pone in stato di attesa mentre detiene una risorsa condivisa.
- waiting circolare: se viene a formarsi una configurazione di waiting in cui ogni thread ne aspetta un altro.

Vediamo adesso come prevenire ognuna delle quattro condizioni di deadlock sopracitate

- risorse limitate: fornire risorse sufficienti. In modo semplice basta aumentare la quantità di risorse disponibili in modo che almeno un thread possa giungere a completamento nonostante gli altri siano bloccati in uno stato intermedio. In tal modo il processo che giunge a conclusione rilascia le sue risorse mettendole a disposizione dei thread in attesa.

- non avere prerilascio: inserire il prerilascio. Ovvero consentire al sistema di reclamare forzatamente le risorse.
- attesa mantenendo una lock: rilasciare le lock ogni volta che viene effettuata una chiamata ad un modulo esterno.
- waiting circolare: ordinare le lock. Cioè di dare un'ordine ben preciso alle lock e nel caso se ne debbano acquisire alcune, farlo sempre nell'ordine prestabilito.

I quattro modi di evitare una deadlock visti finora sono considerati rimedi *statici*. Esistono anche dei metodi *dinamici*, cioè utilizzabili per prevenire una deadlock mentre il programma è in esecuzione. Uno di questi è dato dall'*algoritmo del banchiere*.
Nell'algoritmo del banchiere i thread calcolano la massima quantità di risorse di cui avranno bisogno prima di cominciare ad eseguire un compito acquisendole e rilasciandole durante l'esecuzione. I ritardi provocati garantiscono che il sistema non incorra mai in uno stallo. L'algoritmo divide tutti gli stati in cui si trova un processo in base ai possibili stati futuri di quel processo:

- stati sicuri: sono gli stati in cui per ogni possibile combinazione di richieste future riuscirò a giungere alla fine del compito.
- stati insicuri: è l'insieme degli stati in cui per ogni possibile combinazione di richieste future ne esiste almeno una in cui il sistema va in stallo.
- stati di deadlock: sono gli stati in cui tutti i possibili stati futuri danno luogo a una deadlock.

Notiamo che non necessariamente gli stati insicuri danno luogo ad una deadlock e che anzi da tali stati potrei tornare nell'insieme degli stati sicuri. Il problema è che una pessima gestione delle risorse o una sequenza di scheduling "sfortunata" potrebbe costringere il sistema in una deadlock.
Lo scopo dell'algoritmo del banchiere è quello di mantenere un processo nell'insieme degli stati sicuri. Nella pratica questo avviene sospendendo ogni richiesta che porterebbe in uno stato insicuro.

Consideriamo infine come ripristinare un sistema che incorre in uno stallo. Alcuni sistemi infatti consentono la formazione di deadlock per ripristinare il sistema successivamente. Perché permettere l'esistenza delle deadlock? Solitamente questo avviene perché è difficile prevedere quando una deadlock avverrà; spesso inoltre è computazionalmente costoso. Solitamente allora si procede in vari modi:

- Procedere senza le risorse: può capitare che in un certo momento non si abbiano tutte le risorse per soddisfare un thread. Talvolta allora si decide di far procedere comunque il thread, poiché potrebbe succedere che quando sarà richiesta la quantità massima di risorse dal thread, esse siano state rilasciate da altri thread.

- Riavvolgi e riprova: Questa tecnica prevede di revocare le risorse ad un thread per provare a concludere il processo distribuendo le risorse in un altro modo. Per implementare questa tecnica si utilizzano dei *safe points* da cui il sistema possa ripartire in caso di deadlock. Solitamente la scelta su quale thread privare delle risorse ricade sul più *giovane*, in tal modo qualche thread, probabilmente i più vecchi potrebbero giungere a compimento.

- Rilevare le deadlock: adesso che abbiamo stabilito come gestire le deadlock, dobbiamo occuparci di come rilevarle. Solitamente il metodo più facile di rilevare le deadlock consiste nell'implementare un grafo tra threads e risorse. Se in tale grafo risultano esservi cicli ad un certo stato del processo, allora abbiamo raggiunto uno stato di deadlock.

6.6 Sincronizzazione non bloccante

7 Scheduling

Quando abbiamo compiti multipli da eseguire può essere naturale chiedersi quale compito eseguire per primo. La *politica di scheduling del processore* risponde esattamente a tale domanda determinando quale thread venga eseguito per primo.

7.1 Scheduling uniprocessore

Cominciamo dando alcune definizioni fondamentali per il seguito. Definiamo:

- *workload* l'insieme dei compiti che deve essere svolto da un sistema. Dobbiamo però fare un'importante distinzione tra:
  Compute-bound task: compiti che fanno ampio uso del processore
  I/O-bound task: l'insieme dei compiti che trascorrono buona parte del loro tempo in attesa di input e output sfruttando il processore solo raramente
- *work-conserving policy*: ogni volta che ci riferiamo a compiti eseguiti da processori parliamo di processori work-conserving se non lasciano mai il processore inattivo nel caso vi sia del lavoro da fare
- *prerilascio*: nel seguito ci riferiremo a processori capaci di espropriare un thread delle sue risorse ed affidarle a qualcun altro
- *overhead*: il lavoro "extra" che viene effettuato dal processore per inserire un nuovo thread in esecuzione in sostituzione di quello vecchio

Possiamo ora definire le varie politiche di scheduling. La prima, e più intuitiva, è la first in first out(FIFO). In questa politica i compiti vengono svolti nell'ordine in cui arrivano fino alla terminazione di ognuno. La politica FIFO minimizza l'overhead e apparentemente sembrerebbe essere equa. In certi casi però questa politica risulta essere estremamente inefficiente perché se un insieme di thread che devono eseguire un piccolo compito ognuno giungono al processore subito dopo un altro thread che deve svolgere un compito assai più lungo, il tempo medio di attesa risulterà essere assai elevato.
In modo diverso possiamo affidarci ad una politica Shortest Job First (SJF) in cui vengono eseguiti prima quei thread il cui tempo di esecuzione è minore in modo da avere un tempo medio di attesa assai più basso. Anche questa politica presenta comunque dei difetti perché ad esempio i processi di maggior durata vengono completati in tempi assai più lunghi e si hanno frequenti cambi di contesto che fanno aumentare l'overhead.
Una politica che giunge ad un compromesso tra le due precedenti è la politica di *round robin*. In questa politica si stabilisce un *quanto di tempo* durante il quale viene eseguito un compito da un thread. Se alla fine del quanto di tempo il compito ha terminato, allora non vi è alcuna conseguenza. Se invece, alla fine del quanto di tempo il compito non è stato completato, allora occorre un interruzione che salva i dati riguardo al thread per poi cedere il processore ad

un altro thread. Il thread "interrotto" viene messo in una coda e verrà eseguito nuovamente più tardi. La scelta della lunghezza del quanto di tempo è fondamentale: un quanto di tempo troppo lungo renderebbe la politica round robin simile alla FIFO. D'altro canto se la durata è breve si rischia di avere molti cambi di contesto con alto costo di overhead. In ogni caso anche con una buona scelta del quanto di tempo questa politica non è esente da difetti in quanto se i compiti sono di durata simile, la politica round robin li termina più o meno tutti nello stesso momento dando luogo a tempi medi di attesa più alti di quelli che avremmo avuto usando FIFO o SJB.
Talvolta si decide di implementare algoritmi che rispettino dei criteri di equità. Ad esempio, nel caso in cui un thread esegua un compito di I/O in cui l'utilizzo del processore è marginale, la porzione di tempo che non viene usata da esso viene redistribuita tra i rimanenti threads in modo equo.
Notiamo infine che buona parte dei sistemi operativi moderni ha deciso di utilizzare una politica di scheduling chiamata *Multi-level Feedback Queue* (MFQ). Questa politica è di compromesso, infatti si comporta mediamente bene su ogni sfida di scheduling proposta finora. Possiamo dire che la politica MFQ è un'estensione della round robin infatti opera su diverse code a diversi livelli di priorità. L'idea è di eseguire prima i thread in coda ai livelli più alti e ogni volta che un thread viene eseguito è poi "degradato" ad una lista di priorità inferiore.

7.2 Scheduling multiprocessore

7.3 Energy aware scheduling

7.4 Scheduling in tempo reale

7.5 Teoria delle code

7.6 Gestione del sovraccarico

7.7 Server in un data center

8 Traduzione degli indirizzi

Quello di cui ci interesseremo in questo capitolo è la traduzione degli indirizzi; ovvero la conversione da indirizzo di memoria a locazione fisica. Tramite la traduzione il sistema operativo può: isolare processi, far comunicare i processi tra loro tramite regioni di mermoria condivise, allocare efficientemente la memoria e moltissime altre cose.

8.1 Il concetto di traduzione di indirizzo

Per venire incontro a tutte le necessità elencate poco sopra il meccanismo di traduzione risulta essere abbastanza complesso e, per cominciare dobbiamo dare le definizioni di:
Indirizzo virtuale: è l'insieme di memoria e indirizzi visti dai processi. Tali indirizza sono chiamati virtuali perché non corrispondono necessariamente a un elemento fisico.
Indirizzo fisico: dal punto di vista della memoria esistono solo indirizzi fisici, ovvero locazioni reali della memoria.
Il meccanismo di traduzione serve per conciliare le due visioni contrapposte

8.2 Riguardo la traduzione flessibile

Il meccanismo di memorizzazione *base and bound* presentato nel capitolo 2 è efficiente ma nella realtà i processi sono sparpagliati in memoria per questioni di successive aggiunte e rimozioni in memoria. Tale fenomeno è detto *segmentazione* ed esiste un modo facile per gestirlo: invece che tenere una coppia di base e limite, si tiene un array di coppie di basi e limiti, dove ogni coppia rappresenta un segmento di memoria occupata dal processo. Tramite una soluzione di questo tipo è anche possibile che il sistema operativo assegni a segmenti diversi proprietà diverse, oppure fare in modo che alcuni segmenti siano privati ed altri condivisi con altri processi. Se un programma cerca di accedere in uno degli intervalli di memoria che non gli appartengono (cosa che viene sempre controllata) viene sollevata un'eccezione chiamata *segmentation fault*. Un altro esempio a riguardo è il meccanismo *zero on reference*; poiché a priori non è noto quanto spazio in memoria occuperà un programma, con questo meccanismo si alloca una certa quantità di memoria di cui vengono azzerati solo i primi bit. Se questa quantità viene esaurita viene sollevata un'eccezione e il sistema operativo azzera altri bit. I limiti presentati da questa configurazione risiedono nel fatto che ad un certo punto potrei avere abbastanza memoria libera per accogliere un nuovo processo ma, essendo non contigua, non utilizzabile. Questo fenomeno è chiamato di *frammentazione esterna*.

Per limitare questo fenomeno si utilizza un modello a *memoria paginata*. L'idea è quella di utilizzare delle *pagine* di dimensione fissa, e invece di avere una coppia base e limite ogni processo ha una propria *page table* contenente i puntatori alle varie pagine del processo. Essendo la dimensione fissa anche la gestione di dati

condivisi risulta più semplice e per tenere traccia di quali pagine di memoria condivise sono ancora in uso e quali no, si utilizza la *coremap*.
Uno dei problemi che si presentano nella gestione delle pagine è la dimensione della pagina. Con pagine troppo piccole, la tabella delle pagine risulta più complessa e più dispendiosa nella gestone ma d'altro canto pagine troppo grandi rischiano di restare in buona pate inutilizzate.
Un passo ulteriore che viene fatto nella gestione della memoria è quello della *segmentazione paginata* in cui ogni segmento punta a una page table.
Un altro approccio è quello della *paginazione a più livelli*. Non avremo più una page table con i puntatori alle pagine ma una tabella di puntatori a page table da cui ricavare gli indirizzi delle pagine.
L'approccio utilizzato da x86 consiste nell'avere una *Global Descriptor Table*, equivalente a una segment table, salvata in memoria. Ogni ingresso di tale tabella punta a una tabella delle pagine multilivello.

8.3 Riguardo la traduzione di indirizzi efficiente

In questo paragrafo vedremo come mantenere inalterata la struttura sulla traduzione di indirizzi migliorandone le prestazioni. Per farlo useremo una *cache*, ovvero una copia dei dati a cui avere accesso più rapidamente.
Introduciamo dunque la *translation lookaside buffer*, ovvero una piccola tabella contenente i risultati delle recenti traduzioni di indirizzo. Solitamente l'hardware associato al TLB controlla se l'indirizzo da tradurre è già presente nella tabella controllandone tutti gli ingressi in contemporanea. Se ha successo abbiamo una *TLB hit* altrimenti abbiamo una *TLB miss*. Ovviamente l'hardware della TLB è di modeste dimensioni, scritto su chip di memoria statica e collocato vicino al processore per rendere le operazioni quanto più veloci possibile. Il costo di una ricerca tramite TLB viene solitamente riassunto con l'equazione che segue:

$$\text{Costo(traduzione)}=$$
$$\text{Costo(ricerca nel TLB)}+ \text{Costo (traduzione intera) x (1-P(hit))}$$

dove P(hit) è la probabilità di ottenere un hit nella ricerca nel TLB. Ovviamente dovremmo fare in modo che P(hit) sia più che mai alta. Per farlo si utilizza la *superpagina*, un'insieme di pagine contigue nella memoria fisica. Per distinguerle facilmente dalle pagine si utilizza un bit apposito.
Ovviamente ci sono anche delle problematiche da risolvere quando si utilizzano le TLB: in primo luogo ogni volta che viene fatto un cambi di contesto è necessario fare in modo che il registro della page table punti la page table del nuovo processo in esecuzione, anche per evitare che nuovi processi abbiano accesso ai dati dei processi precedentemente in esecuzione. Un'altro problema è se i permessi di una pagina vengono ridotti; in tal caso il kernel deve assicurarsi che il TLB possieda una copia aggiornata delle traduzioni prima di ripristinare il processo.
Inoltre nelle macchine multiprocesseore, dove ogni processore ha la propria copia

degli indirizzi nel proprio TLB, può succedere che debba essere modificato o rimosso un valore contenuto nella TLB. In tal caso, poiché ogni processore può modificare la propria TLB, solitamente il sistema operativo interrompe le attività su tutti i processori per rimuovere i valori da tutte le TLB. Questa dispendiosa operazione viene solitamente chiamata *TLB shootdown*.

Un'ulteriore passo per sviluppare le prestazioni consiste nell'inserire un'ulteriore strato di cache consultabile prima della TLB contenente direttamente i dati associati all'indirizzo virtuale cercato. Se si ha successo è possibile utilizzare direttamente tali dati. In ogni caso si devono risolvere le stesse problematiche presentate con la TLB.
L'ultima ottimizzazione che viene fatta di frequente nelle macchine è quella di avere una cache fisica di secondo livello, così chiamata perché viene consultata dopo un miss nella cache virtuale e un miss nella TLB, ma prima di cercare l'indirizzo nella memoria fisica.

8.4 Protezione software

9 Caching e memoria virtuale

9.1 Il concetto di cache

Cominciamo introducendo quache termine fondamentale:
Cache hit: si verifica quando, cercando un valore, viene trovato nella cache.
Cache miss: si verifica quando, cercando un valore, non viene trovato nella cache.
Località temporale: affinché la probabilità di avere una hit sia alta ci accorgiamo del fatto che i programmi tendono a riutilizzare le stesse istruzioni e gli stessi dati a cui hanno acceduto di recente.
Località spaziale: allo stesso modo i programmi tendono ad utilizzare dati spazialmente vicini a quelli usati di recente. Infatti si preferisce caricare blocchi interi di dati piuttosto che singole locazioni.
Per quanto riguarda la scrittura su cache esistono due tipi di cache: la cache *write-through* che salva sempre le modifiche fatte nella cache anche nella memoria permanente oppure la cache *write back* che salva le modifiche della cache all'interno della cache stessa per poi modificare gli originali solo quando resta poco spazio nella cache.

Una tecnica utilizzata molto frequentemente è quella del *prefetch*. Poiché l'accesso in memoria è un processo lento rispetto alla computazione, si decide di cominciare a prelevare il blocco di memoria successivo mentre quello corrente è ancora in esecuzione. Il costo di queste operazioni è dato dall'equazione:

$$\begin{gathered}\text{Latenza(lettura)} = \\ \text{P(cache hit) x latenza(cache hit)} + \\ + \text{P(cache miss) x latenza(cache miss)}\end{gathered}$$

9.2 Gerarchia di memoria

Quando si decide di utilizzare una cache su una macchina è necessario avere una vaga idea dei tempi necessari per accedere alle varie tipologie di cache e memorie. L'idea è la seguente: più piccola è la memoria, più può essere rapida. Inoltre più è lenta, più è economica. Questo spiega perché un sistema abbia più di un livello di memoria. Ad esempio, il rapporto tra tempo di accesso in una cache primaria (da 64 kb) e quello di un disco locale (da 1tb) è pari a 10^{-7} ovvero l'accesso al disco è dieci milioni di volte più lento.

9.3 Quando le cache funzionano e quando no

Potremmo adesso chiederci quanto e quando le cache siano utili. In generale tracciando un grafico in cui si pone sulle ascisse la dimensione della cache e sulle ordinate la percentuale di cache hit, notiamo che già per cache di dimensione piuttosto ridotta (2kb) grazie alle proprietà di località temporale e spaziale, la percentuale di hit si aggira intorno al 75%/80%. Considerando sempre questo grafico, il punto di flessione della curva (amichevolmente detto "ginocchio") è

chiamato il *working set* del programma. Fintanto che è raggiunta tale percentuale avremo un alto numero di hit e le prestazioni saranno buone.
Ovviamente la percentuale di hit è variabile nel tempo: ad esempio varia ogni volta che viene aperto un nuovo programma. Dunque ogni volta che il comportamento dell'utente cambia, avendo delle conseguenze sulla percentuale delle hit, abbiamo un *phase change behavior*.
Un altra modellizzazione utile per comprendere l'utilizzo delle cache per i contenuti internet è quella di Zipf.
Nella *Zipf distribution*, usata originariamente per creare una scala delle parole più usate in un testo, ma del tutto adattabile a molti altri modelli, abbiamo che tale frequenza è data da $\frac{1}{k^\alpha}$

9.4 Ricerca nella memoria cache

Giunti a questo punto ci chiediamo come effettuare le ricerche all'interno della memoria cache. Qual è la struttura dati adeguata per fare ricerche nella cache? Lavorando su memorie di piccole e piccolissime dimensioni le nostre scelte sono limitate, infatti ogni scelta che porta ad un overhead rischia di compromettere i vantaggi di avere una cache. Vediamo tre strutture possibili:

- Fully associative: una cache strutturata fully associative controlla contemporaneamente tutti gli indirizzi presenti nella cache allo stesso momento. E' la struttura più efficiente ma è anche la più costosa. Bisogna anche dire che con il crescere della dimensione delle cache diventa sempre più difficile controllare nello stesso momento tutti gli ingressi della tabella.

- Direct mapped: in questo modo la cache è strutturata come una tabella hash. Ovviamente è molto efficiente nel cercare ma decresce di molto la flessibilità e la probabilità di hit visto che se l'hash di un certo indirizzo è già occupato, non posso mettere quell'indirizzo nella cache.

- Set associative: fonde i due approcci precedenti utilizzando una cache divisa in più tabelle hash. Dette $T_1, T_2, .., T_n$ le n tabelle hash e h(ind) l'indirizzo dopo aver utilizzato la funzione di hash, in set associative controllo in parallelo $T_1[h(ind)], .., T_n[h(ind)]$

9.5 Politica di rimpiazzo

Esistono numerose politihe di rimpiazzo degli elementi della cache, i più noti sono:

- Random: per quanto possa sembrare strano, una politica possibile è quella di rimuovere un elemento a caso dalla cache in quanto, spesso il sistema potrebbe non avere il tempo di fare scelte più complesse, o comunque lo spazio impiegato per tenere tali politiche potrebbe essere speso meglio come spazio di cache extra.

- FIFO: con quesata politica le pagine di un programma trascorrono lo stesso tempo nella cache. Uno degli svantaggi della FIFO è che nel caso in cui un programma cicli su una certa quantità di informazioni non contenibili nella cache, dopo un certo periodo di tempo darà luogo a sole miss.

- LRU: la least recently used consiste nel togliere dalla cache l'elemento che non si usa da più tempo. In certi casi è la politica migliore, ma nell'esempio visto per la politica FIFO si comprta esattamente come quest'ultima. Inoltre, a livello hardware è difficile da implementare e si preferisce utilizzarne un'approssimazione che vediamo nel paragrafo successivo.

- LFU: nella least frequently used stabiliamo chi deve lasciare la cache in base a quale blocco di pagine è stato utilizzato meno nell'ultimo periodo di tempo. Sia LRU che LFU cercano di prevedere in parte ciò che accadrà, infatti molti sistemi usano una combinazione di questi due approcci.

Bisogna infine notare che non sempre l'aumento della dimensione della cache implica un aumento del numero di hit. Una inusuale situazione di questo tipo è chiamata *anomalia di Belady*.

9.6 Files mappati in memoria

Un modo per approssimare LRU è quello di utilizzare l'algoritmo dell'orologio, che funziona come segue. Anzitutto bisogna dire che molte architetture prevedono la presenza di un *bit d'uso* per ogni elemento della page table nella cache. Ogni volta che un blocco viene utilizzato, il suo bit d'uso viene posto ad 1. L'algoritmo porta tale valore a 0 dopo un certo periodo di tempo, mentre i blocchi il cui bit d'uso è già a 0 vengono tolti dalla cache. Questo modo di fare è detto *second chance* in quanto sono necessari due passaggi del processore per eliminare un blocco dalla cache. Un ragionamento di questo tipo è generalizzabile ed è solitamente chiamato *k'th chance* facendo in modo che quando un blocco viene utilizzato il suo bit d'uso (o i suoi bit) segnino un valore k che viene deremendato di uno ogni volta che il processore lo controlla.

9.7 Memoria virtuale

10 Gestione avanzata della memoria

10.1 Zero-copy I/O

10.2 Macchine virtuali

10.3 Fault Tolerance

10.4 Sicurezza

10.5 Gestione della memoria a livello utente

11 File systems: introduzione e panoramica

11.1 L'astrazione del file system

Un *file system* è l'astrazione del sistema operativo che fornisce dati persistenti. Questi dati vengono detti persistenti perché vanno perduti solo nel caso in cui vengano esplicitamente eliminati; non devono andare persi neanche se il computer si spegne o si blocca.
Un *file* è una collezione di dati in un file system avente un proprio nome. Si compone di *dati*, ovvero quelli che interessano all'utente e di *metadati* che riguardano il sistema operativo (ad esempio la dimensione del file, la data di creazione..)
Una *directory* è una lista di nomi leggibili e un percorso da ogni nome allo specifico file o alla specifica directory.
La stringa che identifica un file o una cartella è detta *path*.
Se pensiamo alla directory come ad un albero, allora la sua radice è una directory chiamata *root directory*. I path che non cominciano dalla root directory sono interpretati come path dalla directory corrente e sono detti *relative path*.
La mappatura tra un nome e il file è detta *hard link*.
Diciamo inoltre che un *volume* è una collezione di memorie fisiche che formano un dispositivo a memoria logica. Un computer può unire volumi multipli in una unica gerarchia tramite un operazione di *mounting*, ad esempio quando viene collegata una penna usb ad un computer.

11.2 API

Per concretezza abbiamo un insieme di funzioni standard per accedere a file e directory:
Create(): crea un file con i suoi metadati e crea il nome per quel file nella directory.
Link(): crea un hard link per un file esistente. Dopo che ha avuto successo ci sono paths multipli per uno stesso file.
Unlink(): rimuove il nome di un file da una directory, lasciando che vi si acceda tramite un altro link. Se il link che si elimina è l'ultimo -tale controllo viene fatto tramite un contatore- allora il file associato viene eliminato liberando le sue risorse.
Mkdir(): crea una directory.
Rmdir(): cancella una directory.
Open(): serve per accedere ad un file ottenendo il suo file descriptor. Viene usata sempre la open prima delle varie read() e write() per due motivi: intanto i controlli sui permessi e sui paths possono essere fatti solo tramite la open e vengono fatti una volta sola, inoltre quando viene chiamata la open il sistema operativo ottiene delle informazioni utili come l'ID del file, la posizione corrente..
Close(): serve per terminare l'utilizzo di un file ma lascia intatto il file open del file nel sistema operativo.
Seek(): cambia la posizione corrente di un processo di uno specifico file aperto.
Mmap(): serve per creare una mappatura tra una regione della memoria virtuale

del processo e una certa regione di un file. Quando la mappatura è avvenuta si potrà accedere al file tramite una pagina condivisa nella cache del kernel.
Unmap(): elimina la mappatura creata dalla mmap().
Fsync(): ogni volta che un file viene modificato, la modifica non viene salvata subito in modo permanente. la fsync() si assicura di modificare il file permanente.

11.3 Strati software

Quando consideriamo i sistemi operativi, vengono forniti diversi livelli, o strati, di software. Gli strati "superiori" si occupano di API e performance mentre i livelli inferiori dell'accesso ai dispositivi. Vediamo meglio questa distinzione:

- API e performance: con i livelli software superiori ci occupiamo di gestire le funzioni basilari tramite system call oppure tramite le librerie delle applicazioni se vogliamo aggiungere qualche funzionalità particolare. Un altro compito è quello di gestire la cache infatti è molto importante che la cache sia ben gestita poiché le memorie di archiviazione permanente sono estremamente più lente delle memorie principali. L'ultimo compito è quello del prefetch, con il quale si ottimizza ulteriormente la performance. Bisogna però fare attenzione a non seguire una politica di prefetch troppo aggressiva poiché ad esempio ogni blocco precaricato occupa spazio sulla cache; se ciò che viene eliminato dalla cache per fare posto al nuovo blocco ci servisse prima di ciò che abbiamo caricato, sarebbe stato uno spreco. Inoltre il prefetch consuma risorse sull'I/O. Se viene fatto troppo prefetch si rischia di dover far attendere eccessivamente le altre richieste.

- Accesso ai dispositivi: diciamo anzitutto che i *device drivers* hanno il compito di tradurre dall'astrazione ad alto livello allo specifico hardware di una periferica di I/O. In generale per venire incontro alla grande diversità dei dispositivi collegabili a un sistema operativo si rendono necessari i drivers. Ma la presenza degli strati software aiuta nel compito di gestione. Infatti ad esempio il sistema operativo fornisce un'interfaccia standard per leggere o scrivere quantità fissate di dati(blocchi da 512, 2048 o 4096 bytes). Comunque, per gestire l'accesso ai dispositivi si utilizza una *memory mapped I/O* che traccia una mappatura dei registri dei dispositivi collegati all'I/O bus. Questa mappatura si trova nel memory bus.
  Un altro aspetto da considerare è che i computer trasferiscono blocchi di dati, cioè non parole singole, piuttosto qualche kilobytes. Quando un dispositivo utilizza il *direct memory access* o DMA, vengono caricati diversi blocchi dalla memoria interna del dispositivo alla memoria principale del sistema operativo.
  Infine ogni volta che vengono effettuate delle operazioni su un dispositivo è bene sapere come gestirne le richieste. Un metodo è quello del *polling* in cui viene controllato periodicamente il registro di stato del dispositivo ma

poiché i dispositivi di I/O sono più lenti e le comunicazioni vengono effettuate ad intervalli irregolari si preferisce far segnalare eventi importanti tramite le interruzioni.

12 Dispositivi di archiviazione

12.1 Dischi magnetici

I dischi magnetici sono un supporto di memorizzazione utilizzati ampiamente nei computer.
I dischi magnetici si compongono di uno o più *piatti*. Ogni piatto ha due *superfici* una su ogni lato. Quando il dispositivo si accende i piatti sono in costante rotazione sulla spinta di un motore. Il disco viene letto e scritto tramite una *testina*, ve ne è una per ogni superficie. La rotazione dei piatti genera un sottile strato d'aria in movimento che tiene sospesa la testina a pochi nanometri dalla superficie. Per raggiungere tutto il disco si utilizza un *braccio* collegato ad un motore per consentire gli spostamenti.
I dati sono immagazzinati nei *settori*, la cui dimensione tipica è di 512 bytes. L'hardware in fase di lettura deve percorrere l'intero settore, non ne può leggere solo qualche parola. Un cerchio di settori su disco è detto *traccia*. Per archiviare quanti più dati possibile si cerca di ridurre al minimo la dimensione dei settori e delle tracce.
I settori di un disco sono identificati tramite gli *indirizzi logici di blocco* che specificano la superficie la traccia e il settore cui accedere. In generale il tempo di accesso ad un disco è dato da:

$$\text{Tempo di accesso} = \text{tempo di seek} + \text{tempo di rotazione} + \text{tempo di trasferimento}$$

Il temo di seek è il tempo impiegato dal braccio per raggiungere la traccia desiderata. Nei tempi di seek sono compresi anche il tempo per leggere la stessa traccia sulla superficie opposta. Chiamiamo inoltre tempo minimo di seek il tempo necessario per spostarsi da una traccia a quella adiacente e tempo massimo di seek il tempo per spostarsi dalla traccia più interna a quella più esterna.
Il tempo di rotazione è il tempo che si impiega, una volta raggiunta la giusta traccia a raggiungere il giusto settore.
In generale nei dischi moderni il tempo di seek e il tempo di rotazione sono assai più lunghi essedo dipendenti da componenti meccaniche e non elettriche.

Le performance di lettura di un disco possono essere significativamente migliorate a seconda della politica di scheduling del disco che si decide di utilizzare. I primi approcci che vengono in mente sono:
FIFO: del tutto inefficiente perché in caso di richieste in tracce alternatamente interne e esterne il tempo medio è assai alto.
SPTF/SSTF: (shortest positioning time first e shortest seek time first) anche questi due approcci greedy non vanno bene perché intanto non garantiscono la migliore performance e inoltre vi è rischio di starvation per richieste spazialmente lontane.

Si preferiscono allora altri tipi di politiche:
SCAN: consiste nel muovere il braccio dall'esterno all'interno servendo le richi-

este he occorrono per poi servire le richieste che occorrono spostandosi dall'interno all'esterno.
CSCAN: si servono tutte le richieste dall'esterno all'interno per poi riportare il braccio sulla traccia più esterna e ripetere.
R-CSCAN: consiste nel servire le richieste "spazialmente vicine" rispetto alla richiesta corrente sia in relazione alle tracce che ai settori. Infatti il braccio che si sta spostando dall'esterno all'interno consente piccoli spostamenti nella direzione opposta se necessario.

12.2 Archiviazione flash

13 Files e directories

13.1 Panoramica sull'implementazione

Quando ci si chiede come dovrebbe essere implementato un file system ogni sistema operativo fa a modo suo ma esistono alcune idee chiave che vengono sempre tenute in considerazione:

- Directories: i sistemi operativi convertono il nome di un file in un numero legato al file.
- Strutture indicizzate: una volta che il nome del file è divenuto un numero si utilizza una struttura indicizzata permanente per localizzare i blocchi del file.
- Mappa dello spazio libero: i file systems implementano una mappa per conoscere quali blocchi sono liberi e quali no. Quanto meno un file system deve consentire l'espansione di un file se vi è memoria disponibile, ma poiché la località spaziale è importante i moderni sistemi operativi cercano di trovare blocchi liberi nelle vicinanze dei blocchi del file.
- Euristiche sulle località: poter localizzare sempre i files, non importa dove siano immagazzinati consente di seguire varie politiche per decidere dove un blocco o un file dovrebbe essere archiviato. Una di queste consiste nel deframmentare periodicamente le memorie di massa per avere sequenze contigue libere di maggiore dimensione.

13.2 Directories: dare nome ai dati

Per convertire il nome di un file in un numero si utilizza un algoritmo ricorsivo la cui base è un valore standard associato alla root. Ad esempio il file system di UNIX, FFS utilizza il numero 2 come indirizzo di root.
Comunque, come già visto il sistema operativo consente ad un file di avere più di un nome. Allora gli *hard links* sono i multipli percorsi per giungere ad uno stesso file (cioè al suo numero).
Chiamiamo invece *soft links* quei collegamenti che non giungono al file, ma giungono al suo nome. Collegano dunque un nome ad un altro nome.

13.3 Files: trovare i dati

Vedremo adesso diversi esempi di file systems.

Il primo che vediamo è il file system FAT (file allocation table). Implementata alla fine degli anni 70 e miglioriata successivamente oggi viene usata come FAT-32 che supporta volumi fino a 2^{28} blocchi e file fino a $2^{32} - 1$ bytes.
La tabella utilizzata nella FAT si compone di array a 32 bit in un'area riservata del volume. Ognuno dei valori contenuti in tabella punta al successivo elemento della tabella oppure ad uno speciale valore che segnala la fine del file. La FAT

possiede un elemento per ogni blocco del volume: se l'i-esimo ingresso in tabella è la j-esima parte di un file, allora nella memoria al blocco i si trova il j-esimo blocco di dati del file. Per segnalare che il k-esimo blocco è libero FAT[k] contiene 0.
In ogni caso FAT presenta diversi problemi: necessita di periodica deframmentazione visto che utilizza strategie di allocazione molto semplici. L'accesso casuale non è sviluppato visto che si deve utilizzare un accesso sequenziale ai blocchi. Vi sono pochi metadati e non supporta gli hard links. Infine abbiamo delle limitazioni sulla dimensione dei volume e dei files, infatti 4 dei 32 bit sono riservati dunque con blocchi di 4kb ho $2^{28}x2^{12} = 2^{40} = 1$TB e poiché la dimensione in byte di un file è memorizzata con una stringa da 32 bit, allora la dimensione massima di un file è 2^{32} bytes, ovvero 4gb.

Il secondo che vediamo è lo UNIX FFS (fast file system) basato su un indice multilivello. Ogni file è associato ad un albero la cui radice è detta *inode*. Tutti gli inode sono contenuti in un array e ogni inode è diviso in 16 campi strutturati come segue:
Il primo campo di dimensione fissata è dedicato ai metadati, solitamente ha dimensione 4kb.
I successivi dodici campi sono puntatori diretti a blocchi di memoria. Supponendo che ogni blocco sia di 4kb supportano 48 kb.
Il quattordicesimo campo contiene un puntatore indiretto di primo livello a 1024 blocchi, arrivando a supportare 4mb (1024 blocchi da 4kb l'uno).
Il quindicesimo campo contiene un puntatore indiretto doppio -ovvero punta a 1024 blocchi ognuno dei quali punta a 1024 blocchi- dunque sono indirizzati 1024^2 blocchi, supportando 4gb (1024^2 blocchi da 4kb l'uno).
Il sedicesimo campo contiene un puntatore indiretto triplo che indirizza in tutto 1024^3 blocchi supportando fino a 4tb(1024^3 blocchi da 4kb l'uno).
Per sapere quale inode della tabella bisogna leggere, ad ogni file è associato un *inumber* che indica la locazione da leggere.
La struttura asimmetrica dell'albero lo rende adatto a supportare file di dimensioni diverse, inoltre per stabilire se un blocco sia libero si utilizza una bitmap con un bit per ogni blocco. Per diminuire i tempi di seek FFS divide il disco in gruppi di blocchi contenenti informazioni simili (ad esempio il gruppo di blocchi liberi, il gruppo di blocchi contenenti inode, il gruppo di blocchi di dati..)in zone adiacenti. Infine, per aumentare le prestazioni FFS riserva una parte del disco -circa il 10%- per poter mantenere la struttura a gruppi di blocchi. In questo modo FFS sacrifica una parte di hardware che si è evoluta molto negli ultimi anni per aumentare le prestazioni di componenti hardware che invece hanno avuto un minor margine di crescita.

L'ultimo approccio che vediamo è quello NTFS (new technology file system). A differenza di FAT e FFS, dove l'unità base della memoria è il blocco di dimensione fissata, in NTFS si utilizzano *estensioni* di memoria di dimensione variabile. Ogni file in NTFS è rappresentato da un albero di profondità variabile. Anche se il file è di grandi dimensioni l'albero non è troppo profondo,

fatto che avviene solo se il file è frammentato male.
La radice di tutti questi alberi è situata nella *master file table* o MFT, un array contenente *MFT record* di dimensione 1kb.
La MFT record contiene:
Informazioni standard: sono le informazioni necessarie ad ogni file come la data e l'ora di creazione, di modifica, l'ID del proprietario..
Nome del file:contiene il nome e il file number della sua parent directory.
Lista degli attributi: si utilizza nel caso in cui serva più di un singolo MFT record per contenere un file. In tal caso la lista degli attributi indica quale MFT record contiene quali attributi (ovvero punta alle altre MFT records).
Dati: il contenuto di questa parte è variabile a seconda della dimensione del file. Se il file è piccolo i suoi dati sono contenuti in questa parte; se invece il file è di maggiori dimensioni allora in questa sezione della MFT troviamo i puntatori alle estensioni di memoria che contengono il file.
Per controllare il fenomeno della frammentazione NTFS si riserva come FFS una percentuale di spazio (circa il 12,5%), quando il dispositivo si è riempito, si dimezza lo spazio riservato e si procede iterativamente in questo modo. Inoltre, sempre per evitare la frammentazione il sistema operativo microsoft fornisce uno strumento per deframmentare il disco in modo automatico.

13.4 Accesso ai files e alle directories