Sistemi Operativi 2

# Periferiche I/O

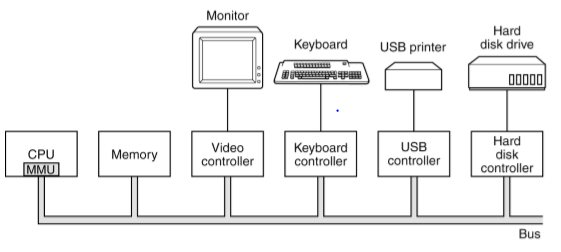
### Principii Hardware dell’I/O

#### Tipi di dispositivi I/O

Le periferiche di Input/Output sono componenti hardware il cui compito è quello di inviare/ricevere segnali dal calcolatore e viceversa, Essi sono suddivisi in due grandi famiglie:

* I dispositivi a blocchi archiviano informativi in blocchi di dimensioni fisse (ognuno col proprio indirizzo), essi possono essere letti/scritti indipendentemente da tutti gli altri. Tutti i trasferimenti sono in unità di uno/più blocchi interi consecutivi;
* I dispositivi a caratteri, a differenza dei precedenti, non possiedono una struttura a blocchi (quindi non sono indirizzabili e non sono accessibili in modo indipendente) e operano su sequenze di caratteri (utilizzando operazioni get e put).

#### Il controller delle periferiche

La periferica, per essere gestita dal sistema operativo, monta una componente elettronica chiamata controller (o adattatore), posto nella periferica o direttamente sulle schede come la motherboard, esso si interfaccia al bus fornendo alla CPU dei registri su cui operare. La comunicazione tra periferica e bus avviene a bassissimo livello. La comunicazione tra CPU e periferica avviene tramite un’interfaccia e può avvenire in due modi:

* Il controller saldato sulla periferica si connette a un’interfaccia che lo espone al relativo bus (PCI per le periferiche veloci/memoria principale e ISA per quelle lente);
* Il controller saldato sulla motherboard espone un’interfaccia alla periferica permettendo così la connessione al bus.

Una volta che la periferica è connessa, l’interazione completa avviene quando La CPU invia al controller stesso dei comandi. Un controller di ogni dispositivo comprende registri che possono ospitare dati o comandi, tra essi vi sono il control register, il data register e il command register: il primo contiene informazioni riguardanti lo stato della periferica verso la CPU, il secondo permette di contenere i dati che la CPU riceve o che deve inviare mentre l’ultimo permette l’esecuzione di determinati comandi da parte della periferica (scrivendo dei flag per segnalare se l’operazione è avvenuta con successo o meno) su ordine della CPU.

### 

### Comunicazione tra CPU e controller

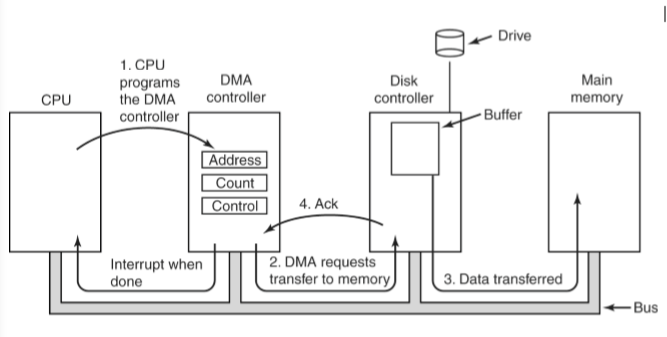
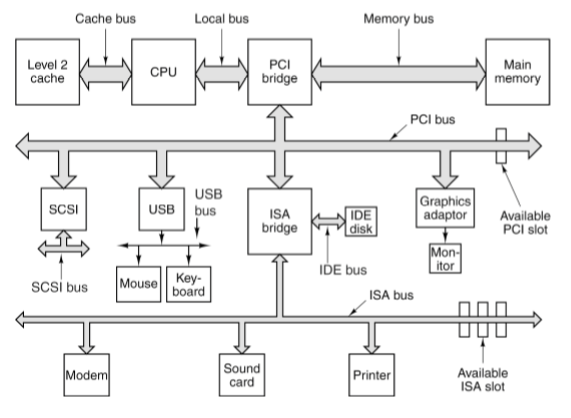
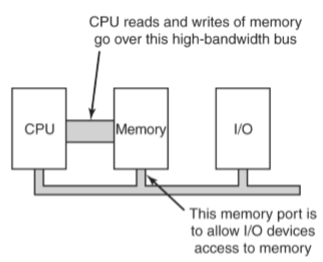
#### Porte di I/O

La lettura/scrittura in questi registri avviene tramite le cosiddette porte di I/O, ognuna assegnata a un determinato registro. Con questa tecnica vengono definiti due spazi di indirizzamento distinti (accessibili solo dal sistema operativo), il primo dedicato alla memoria centrale mentre il secondo dedicato alle porte, al momento della lettura/scrittura posso far riferimento a un indirizzo di una cella di memoria oppure a una porta. L’assegnazione porta-registro viene fatta dal BIOS all’accensione, esso ha il compito di scoprire quanti registri vi sono e, di conseguenze, assegnare delle porte di I/O libere a essi. L’unico problema di questa tecnica è l’utilizzo di un Instruction Set più ricco che, di conseguenza, rende più complessa la CPU.

#### Memory Mapped I/O

Un’altra tecnica è il memory mapped I/O in cui memoria e registri di controllo vengono mappati nello stesso spazio di indirizzi (in generale, gli indirizzi dei registri dei controller si trovano nella parte superiore).Questa tecnica permette di sviluppare del codice utilizzando le stesse instruction set per la lettura/scrittura in memoria (la differenziazione è gestita direttamente dall’hardware) e senza istruzioni speciali fornite dall’hardware, semplificando lo sviluppo dell’applicazione. In più, vi è maggior protezione perché rende non indirizzabile l’area di memoria ai programmi degli utenti (l’area non viene mappata e, di conseguenza, i processi non possono accedervi). Ultima ma non meno importante è la possibilità di manipolare i dati dei registri contenuti in memoria con tutte le istruzioni che manipolano quest’ultima, ad esempio: bisogna creare del codice in cui si deve verificare se una periferica è pronta o meno ad accettare un comando ed eventualmente inviarlo.

| LOOP: IN REG,10000  TEST REG  BEQ READY  BRANCH LOOP  READY: … | LOOP: TEST PORT\_4  BEQ READY  BRANCH LOOP  READY: … |
| --- | --- |

I due codici scritti sopra eseguono il polling rispettivamente sul registro e sulla porta di una periferica che salterà a READY quando sarà pronta a ricevere un comando per poi eseguire le operazioni richieste. In questo modo, però, la cache diventa inconsistente perché la CPU va a prendere dalla cache i valori di PORT\_4 e REG (i quali sono stati inseriti alla loro prima referenza) senza più interrogare il dispositivo e ,di conseguenza, causando un ciclo infinito. Per evitare ciò, l’hardware deve essere in grado di disabilitare la cache in modo selettivo, questa soluzione però renderebbe complesso sia il sistema operativo che l’hardware stesso. Un altro problema è, nel caso sia presente un solo spazio di indirizzi, sia i moduli di memoria che i dispositivi devono testare tutti gli indirizzi per verificare a quali bisogna rispondere, l’ideale sarebbe che gli indirizzi relativi alla memoria fossero testati solo da essa, lasciando tutti gli altri ai controller. Un’altra possibile soluzione è inserire un filtro sul memory bus in modo da smistare gli indirizzi verso il bus corrispondente, esso però sarebbe lento o troppo costoso per l’hardware, di conseguenza si è deciso di sostituirlo con un PCI bridge, un componente, programmato a ogni boot della macchina (questo perché i dispositivi potrebbero cambiare ogni volta), in cui vi sono le associazioni tra un range di indirizzi e i relativi bus di destinazione. Tutto questo però ha un costo: l’hardware diventa più complesso a causa di questo nuovo componente e lo stesso vale anche per il software perché bisogna programmare il BIOS per l’inserimento di tutte le associazioni e dei bus di destinazione. Tra memoria e CPU vi è il già citato memory bus, ovvero il bus specifico per la comunicazione tra le prime due entità, esso è molto più veloce d’un bus ordinario dato che la CPU deve processare continuamente i dati. Se ci fosse un bus ordinario (condiviso tra tutti i dispositivi) al posto del memory bus, la comunicazione tra CPU e memoria sarebbe lenta e andrebbe a concorrere con quella dei dispositivi per il trasferimento dei dati al processore. Le operazione I/O nei PC sono solitamente maggiori rispetto a quelle in memoria, quindi conviene gestirle in modo efficiente. Per farlo, si può utilizzare un’interazione basata sugli interrupt in cui, alla ricezione di un interrupt, la CPU interrompe quello che stava facendo per eseguire codice relativo all’interrupt stesso per poi tornare allo stato precedente quando ha finito. Tutto questo è possibile grazie a un particolare registro della CPU: l’interrupt vector, esso è un array grande quanto il numero di linee connesse che, alla ricezione di un interrupt, salta al relativo codice. In seguito si passa alla gestione degli interrupt attraverso due context switch. Tuttavia vi è un problema: gli interrupt sono molteplici e questo rallenterebbe di molto il resto delle attività della CPU, quindi bisogna in qualche modo gestire questo traffico, come fare? Si introduce Il Direct Memory Access.

#### Direct Memory Access (DMA)

Il DMA è un componente che ha il compito di gestire tutti i trasferimenti e gli interrupt intermedi svegliando la CPU una volta completati. Tutto questo avviene perchè la CPU interagisce col DMA dicendo su quale dispositivo operare, quanti byte leggere e in quali locazioni salvarle, scrivendoli nei suoi registri. Questo meccanismo si chiama Fly By Mode perchè il DMA scrive direttamente in memoria, in assenza di esso bisogna utilizzare un ulteriore accesso al bus per scrivere le parole nel posto desiderato, nonostante sia più flessibile.

#### Accesso al bus

L’accesso al bus avviene nei seguenti modi:

* Cycle Stealing: Viene rubato occasionalmente un ciclo alla CPU, rallentandola leggermente. E’ una tecnica inefficiente anche se va bene per il trasferimento di pochi byte;
* Block mode: Il DMA comunica a un dispositivo di acquisire un bus per blocco (non per byte), inviare dei trasferimenti e infine lo rilascia. Questa tecnica è più efficiente rispetto a quella precedente perché permette il trasferimento di più parole con una sola acquisizione del bus, però si può involontariamente bloccare la CPU e altri dispositivi nel caso in cui vi siano trasferimenti di grossa entità.

#### Interazione hardware tra CPU e periferiche

L’interazione tra dispositivi I/O e CPU avviene attraverso gli interrupt, essi sono segnali che vengono generati dai controller e, alla ricezione, il processore comunica con la periferica per la gestione dell’interruzione. Tutto questo viene gestito da un componente detto interrupt controller il quale, al rilevamento di un interruzione, segnala l’avvenimento alla CPU che, dopo aver chiesto di quale dispositivo si tratta, manda in esecuzione la procedura corrispondente. Ogni dispositivo ha una parte di codice che permette la gestione delle proprie interruzioni, l’interrupt handler: essi vengono caricati in aree di memoria non accessibili agli utenti a ogni avvio del sistema dal BIOS. Alla segnalazione di un interrupt, si recupera l’indirizzo della prima istruzione della relativa procedura, lo si carica nel Program Counter e viene fatto un context switch, sospendendo quindi il processo in esecuzione. Questo tipo d’interazione, però, non è esente dai problemi: Si è costretti ad avere un numero limitato di linee nella CPU e, nel caso dei processori vecchi in cui vi sono poche linee, ognuna può essere associata a più device e questo può portare a problemi. In più, il sistema operativo deve sapere la corrispondenza tra periferica e interrupt handler dato che quest’ultime non si trovano in locazioni di memoria predefinite. Per risolvere questo problema si utilizza l’interrupt vector, una struttura hardware che contiene gli indirizzi della prima istruzione di ogni procedura.

### Principii software per l’I/O

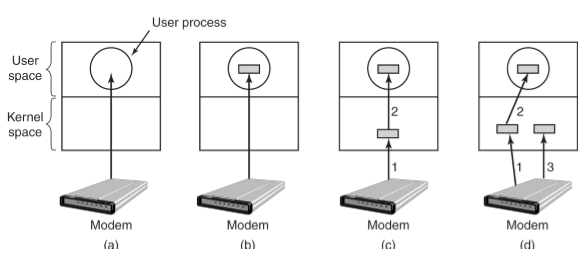
#### Device Indipendence e Naming Uniforme

La parte di codice inerente ai dispositivi I/O deve innanzitutto essere indipendente dal device stesso, ovvero che il software deve svolgere ogni sua funzione indipendentemente dal device cui accede utilizzando le API (Application Programming Interface), delle interfacce rispettate dai driver che rendono omogenea la gestione delle periferiche da parte dell’utente. La differenziazione d’ogni comando è a carico del sistema operativo stesso, il quale risolve tutti i problemi riguardanti la diversità dei dispositivi. Strettamente correlata al primo punto è l’utilizzo d’uno schema di naming uniforme, esso permette di differenziare i singoli device attraverso un nome, il quale è indipendente dal device stesso. In questo modo sono referenziabili per le operazioni di I/O e indirizzabili allo stesso modo. Ogni dispositivo viene identificato attraverso due numeri: il major number e il minor number, il primo identifica il tipo di periferica mentre il secondo la relativa istanza.

#### Gestione degli errori

La gestione degli errori è un’altra parte importante, anche se in generale è preferibile farlo a livello hardware (ad esempio, nel caso in cui un controller scopre un errore in lettura, esso dovrebbe correggerlo, se possibile), quando un errore non è risolvibile a basso livello, la gestione software degli errori entra in azione attraverso i driver. Gli errori possono essere transienti, cioè che sono emersi a causa di un’anomalia temporanea e si risolvono da soli, oppure consistenti, ovvero quando sono causati da un difetto fisico della periferica oppure non sono risolvibili, nel caso in cui si manifestano un errore consistente, il software deve riportare l’applicativo alla situazione precedente (anche abortendola).

#### Sincronia delle Operazioni

Un’ altro aspetto riguarda la sincronia delle operazione: la CPU gestisce l’I/O in modo asincrono però, per semplicità, i programmi utente utilizzano operazione di I/O sincrone ( si bloccano fino a quando non vi è una richiesta da parte dell’utente),quindi sta al sistema operativo gestire in modo trasparente le operazioni asincrone in modo da non farle sembrare tali all’utente.

#### Buffering

Un altro problema riguarda il buffering, ossia l’utilizzo di buffer ( aree di memoria temporanea in cui preservare i dati prima di consumarli) per la riconciliazione di due o più dispositivi che vanno a velocità differenti. La creazione di un buffer a livello utente, però, comporta un grosso impatto nelle prestazione dell’I/O (a causa delle operazioni di copia) e anche una programmazione più complessa. Per risolvere quest’ultimo problema, si crea un buffer a livello kernel in cui vengono svolte le operazioni date e il sistema operativo si preoccupa di copiarlo a livello utente. Dal momento che i buffer sono soggetti a modifiche, la soluzione migliore è utilizzare un pool di buffer a livello kernel in cui il sistema operativo si preoccupa di copiarli a livello utente quando necessario. La soluzione migliore sarebbe utilizzare un buffer per ogni applicazione, questo però richiederebbe una quantità maggiore di RAM, come fare in tale caso? Si utilizza un pool di buffer multiplexati a dovere a ogni applicazione.

#### Gestione delle periferiche condivibili/non condivisibili

Ultima ma non meno importante è la gestione delle periferiche condivisibili e non, esse infatti devono essere gestite dal sistema operativo in modo da evitare problemi. La gestione delle periferiche non condivisibili, in particolare, deve evitare l’insorgere di deadlock durante il funzionamento.

#### Interazione software tra CPU e periferiche

l’interazione tra CPU e periferiche nella parte software avviene in tre modi: I/O programmato, quello guidato dagli interrupt e quello con DMA. Nell’I/O programmato, Il sistema operativo interagisce con la periferica in modo diretto, utilizzando l’attesa attiva per evitare i costi delle interruzioni e anche perché queste ultime richiedono un tempo maggiore rispetto a quello della periferica stessa. Alla chiamata di una system call, il sistema operativo copia i parametri dati nella parte kernel per ragioni di protezione: un segmentation fault in modalità kernel risulterebbe fatale per la macchina stessa. Questa tecnica prevede che il kernel controlli continuamente se la periferica sia libera e, nel caso lo fosse, la fornisca di comandi. Nonostante questa tecnica sia piuttosto semplice, essa ha lo svantaggio di tenere occupata la CPU fino al termine dell’I/O.

| CopyFromUser(buffer, p, count);  /\* il buffer è quello dell’area utente,  p è l’area del kernel in cui viene copiato  e count è il numero di byte \*/  for i=0 to count  while(\*printer.statusReg!=READY);  \*printer.dateReg=p[i];  ReturnToUser(); |
| --- |

L’I/O guidato da interrupt utilizza un ragionamento duale rispetto alla prima tecnica: La CPU fa qualcos’altro e viene chiamata dalla periferica attraverso un interrupt, in questo modo si evita l’utilizzo dell’attesa attiva. Tuttavia la gestione degli interrupt potrebbe avere tempi più lunghi rispetto a quelli di gestione della periferica stessa, come fare in questi casi? Si può cambiare la tecnica a seconda della situazione: se il tempo di gestione del dispositivo supera quello degli interrupt, utilizzo questi ultimi aggiornando driver e interrupt handler, altrimenti continuo con l’attesa attiva. Ogni volta che la periferica stampa un carattere, l’interrupt handler controlla se ce ne sono altri, se non ce ne sono si sblocca l’utente, altrimenti si procede col successivo.

| CopyFromUser(buffer,p,count);  EnableInterrupt();  while(\*printer.statusReg!=READY);  \*printer.dateReg=p[i];  Scheduler(); | if(Count==0) Unblock.user();  else \*printer.dataReg=p[i++];  count--;  Ack.Interrupt();  ReturnToUser(); |
| --- | --- |

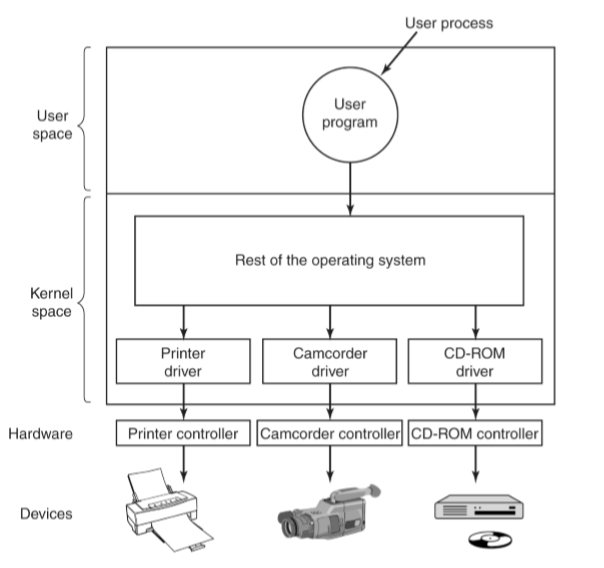
L’I/O con DMA è un caso particolare della prima tecnica in cui il DMA stessa “fa da processore” utilizzando l’attesa attiva senza disturbare la CPU, in questo modo quest’ultima può fare altre cose durante la gestione dell’I/O. Tuttavia rimane il fatto che il DMA non è una CPU, essendo genericamente più lenta, in questi casi allora conviene utilizzare una delle due tecniche citate prima.

| CopyFromUser(buffer,p,count);  SetDmaController();  Scheduler(); |
| --- |

### Gestione degli interrupt

gli interrupt sono segnali elettrici che permettono alla CPU di svolgere codice dedicato alle periferiche, esse devono il più nascoste possibili dal sistema operativo in modo che quest’ultimo ne sappia il meno possibile, per farlo si utilizzano i driver che, al verificarsi di un’interruzione, fa partire un operazione I/O.

### Driver

I driver sono software che si interfacciano direttamente ai controller delle periferiche, esportando un’interfaccia delle operazione al sistema operativo. Il sistema operativo per poter utilizzare l’interfaccia deve essere a conoscenza di tutte le operazioni messe a disposizione dall’interfaccia e, dato che ognuna di esse è diversa dalle altre, bisognerebbe aggiornare il sistema operativo per ogni driver nuovo o aggiornato (prima foto). Come fare per evitare tutto questo? Si fa in modo che i driver esportino tutti la stessa interfaccia e questo non è un problema perchè il sistema operativo sa quali funzione ha a disposizione per ogni dispositivo (seconda foto). Il caricamento dei driver può avvenire staticamente oppure dinamicamente: nella prima soluzione i driver vengono compilati assieme al sistema operativo stesso, questa soluzione però è obsoleta perché sarebbe necessario compilare tutti i driver di tutti i dispositivi (anche quelli non utilizzati) con la conseguenza di ottenere un codice eseguibile enorme. In più la presenza di più driver per uno stesso tipo di periferica renderebbe complicato al sistema operativo stabilire una prelazione. Per questo motivo si è deciso di passare a un approccio dinamico in cui ogni driver è un modulo compilato e richiamato solamente quando necessario, essi contengono tutte le informazioni sulle periferiche che gestiscono. Al collegamento di una periferica, il sistema operativo carica il modulo corrispondente nel kernel (e lo rimuove quando non serve) mentre esso è in esecuzione, il tutto senza il bisogno di riavviare il sistema. In questo modo non si spreca codice eseguibile e in più è più facile sviluppare driver per ogni nuovo dispositivo senza dover ricompilare tutto il sistema operativo, basta solamente l’interfacciamento a esso. 

Il funzionamento generico di un driver è simile al seguente pseudo-codice:

| while(1)  riceviComando();  verificaParametri();  if(BUSY) accodaComando();  else eseguiComando()  attendiCompletamento() |
| --- |

In poche parole, il driver si blocca fino a quando non vi è qualcosa nel buffer, quando sbloccato controlla la correttezza del comando nel buffer, lo traduce per essere leggibile dal controller e, se il buffer è pieno, la mette in coda insieme alle altre operazioni, altrimenti la invia, aspetta il completamento e gestisce l’output. Lo pseudo codice dettato prima, però, non va bene perchè vi è la necessità di avere due flussi di controllo distinti, il primo si preoccupa di mettere in coda le richieste del sistema operativo mentre il secondo le esegue nell’ordine dato.

| LETTORE:  while(1)  OttieniProssimoComando(buffer);  VerificaCorrettezza(buffer);  TraduciComando(buffer,output);  Invia(operazione);  AttendiCompletamento()  GestisciOutput() | SCRITTORE:  while(1)  RiceviComando(buffer);  Accoda(operazione); |
| --- | --- |

Durante la gestione dell’output potrebbero arrivare ulteriori interrupt, essi verranno gestiti in uno dei seguenti modi:

* Se il dispositivo che si sta gestendo è indipendente all’utente, vuol dire che l’interrupt proviene da un dispositivo diverso e quindi non succede nulla, ci sarà comunque un context switch dato che il driver è scritto in modo che il codice sia rientrante: le istruzioni non lasciano mai uno stato inconsistente e quindi vengono eseguite in modo atomico.
* Se il dispositivo dipende dall’utente, si disabilitano le interruzioni.

#### Device hot-plug e plug & play

I device hot-plug sono dispositivi che possono essere aggiunti/rimossi senza il bisogno di riavviare la macchina, di conseguenza il codice deve prevedere e gestire le situazioni in cui l’utente agisce in modo imprevisto. Il plug & play é una componente del sistema indipendente dai driver che rileva l’aggiunta/rimozione di un dispositivo segnalandolo al relativo driver.

### Dimensioni dei blocchi

La dimensione dei blocchi viene scelta in modo indipendente dal dispositivo, in questo modo è possibile dare astrazione a livelli software più alti.

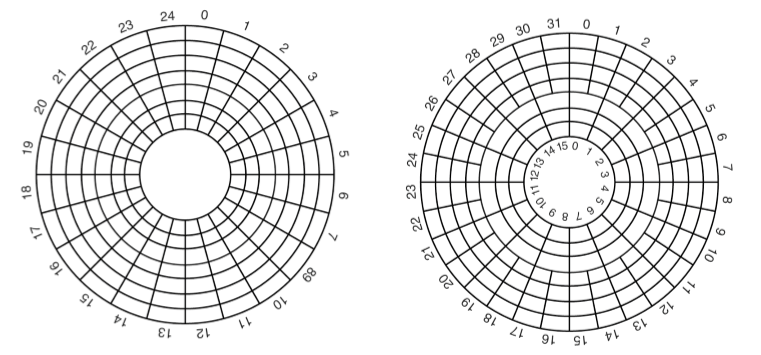
### 

### I/O in modo utente

Questo livello software è esterno al kernel e permette l’interfacciamento con quest’ultimo fornendo le system call. Queste funzioni però non vengono richiamate direttamente dall’utente, esse infatti sono contenute in librerie di funzioni richiamabili dall’utente quando necessario. Un’altra tecnica è lo spooling e consiste nell’interagire adeguatamente coi dispositivi che funzionano in modo esclusivo nei sistemi multiprogrammati, la sua assenza infatti permetterebbe a un processo di bloccare l’accesso a un device a tutti gli altri anche se non lo sta effettivamente utilizzando. Lo spooling funziona utilizzando un processo daemon e una directory speciale detta directory di spooling, in questa cartella i processi inseriscono tutti i file da inviare al device. Il processo daemon infine creerà una coda contenente tutti i file della directory di spooling e li invierà al dispositivo interfacciandosi con esso. In questo modo si evita che i processi occupino un dispositivo a lungo senza che sia necessario.

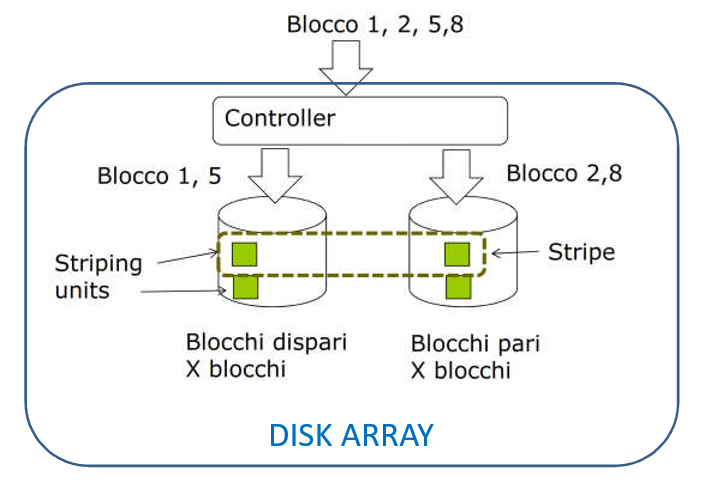
### I Dischi

I Dischi sono dispositivi a blocchi che permettono di memorizzare consistentemente i dati e tutto quello che permette il funzionamento del sistema operativo, essi sono formati da piatti, impilate su un asse, aventi uno strato magnetico su entrambe le superfici. La lettura/ scrittura avviene mediante delle testine in cui , dato l’indirizzo, vengono svolte le operazione date sul settore indicato, esse sono tutte montate sul cosiddetto braccio, il quale permette di spostarle. Ogni piatto è diviso in tracce (i cerchi concentrici) a loro volta divise in settori, un cluster è formato da più settori appartenenti a una traccia. Per individuare un settore all’interno del disco, si utilizza una terna di numeri, la cosiddetta CHS (Cylinder Head Sector): il primo numero si riferisce alla traccia, il secondo alla testina mentre il terzo al settore da leggere. La prima configurazione di CHS prevedeva 10 bit per C (numero di tracce per cilindro), 8 bit per H (il numero di testine) e 6 bit per S (numeri di settori per traccia). La dimensione del disco è data dal prodotto tra la dimensione del singolo settore per 2^24, ovvero il numero massimo di settori della configurazione CHS data precedentemente.



Col progresso tecnologico si è deciso di estendere CHS da 24 a 28 bit, rispettivamente 16 per C, 4 per H e 8 per S, l’unico problema è che lo standard dei BIOS era differente e, di conseguenza, vi era una strozzatura nel disco. Come procedere? Si cambia schema di indirizzamento, passando da CHS a LBA (Logical Block Addressing): L’indirizzamento di questo standard è lineare e permetteva la rappresentazione di 2^20 indirizzi in totale, con l’avanzamento tecnologico sono state intraprese due differenti strade: la prima è l’aumento della dimensione dei blocchi a 4kB rispetto ai 512 byte precedenti mentre la seconda estende LBA rendendolo a 48 bit. La descrizione del numero di piatti, cilindri e settori è la cosiddetta geometria del disco. Dato che la velocità lineare verso l’esterno aumenta con la velocità del disco, per ridurre le mari meccaniche si è deciso di utilizzare una configurazione in cui le tracce sono più fitte verso l’esterno, dato che lo spazio fisico attraversato dalla testo è maggiore rispetto all’interno dello stesso intervallo di tempo. Il tempo di impiegato per spostare la testina sulla traccia contenente il settore desiderato è il cosiddetto tempo di seek, esso varia tra i 3 e i 12 millisecondi a seconda della fascia di qualità. La latenza rotazione è il tempo per arrivare a una traccia dato il settore, essa è dipendente dalla velocità del disco.

La velocità tra RAM e disco, però, rimane ancora abissale. Una possibile soluzione può essere la sostituzione dei dischi magnetici con degli SSD, però vi è ancora differenza di velocità con la RAM, come fare i questi casi? Si utilizza un disk Array, una batteria di dischi che funzionano come se fossero un unico grasso disco, in essa è possibile definire i seguenti concetti:

* Stripe: blocchi sulla stella linea;
* Striping Unit: blocchi singoli.

Come miglioramenti ho l’utilizzo di più dischi in contemporanea e, in più, tutto diventa più veloce, soprattutto quando si è sulla stessa stripe. L’unico problema rimane l’affidabilità, infatti vi è la probabilità che uno dei dischi si rompa e, di conseguenza, non funziona più niente, come fare in questi casi? Si utilizza la ridondanza, ovvero si “sacrifica” parte della capacità reale del disk array per utilizzarla per tenerci copie di dati specifici, verranno utilizzati quando uno dei dischi si rompe, in questo modo tutto continua a funzionare anche se si rompe un disco. Questo metodo è chiamato RAID (Redundant Array Of Independent Disk). Le dimensioni delle striping unit sono molto importanti perché influenzano la velocità di trasferimento dei dati, infatti più sono piccole, più dischi coinvolgo e più vado veloce, tuttavia ho bisogno di accodare le richieste nel caso in cui avvengano simultaneamente.

### Istanze RAID

#### RAID 0 - Striping puro

Nel RAID 0 viene utilizzato lo striping, in pratica la memoria viene divisa in blocchi e ognuno di essi verrà assegnato al rispettivo disco utilizzando l’operazione di modulo, essa permette di aumentare la velocità dato che il controller può leggere/scrivere in parallelo più settori in dischi differenti. Questa istanza permette di ottenere le massime prestazioni dato che la capacità utile equivale a quella massima, tuttavia non è affidabile: basta solamente la rottura di un singolo disco affinché tutto smetta di funzionare.

#### RAID 1 - Mirroring puro

Il RAID 1 è il caso opposto rispetto a quello precedente: vi è la massima ridondanza al costo di una capacità utile equivalente a metà di quella totale, infatti metà dei dischi sono quelli effettivamente utilizzati mentre tutti gli altri sono la copia esatta dei primi. In lettura le prestazioni sono buone nel caso bisogna leggere intere strip dato che è possibile farlo in parallelo. In scrittura la situazione è differente: dato ogni disco ha un suo clone, bisogna ripetere ogni operazione due volte e in più queste si accodano e devono essere scritte contemporaneamente.

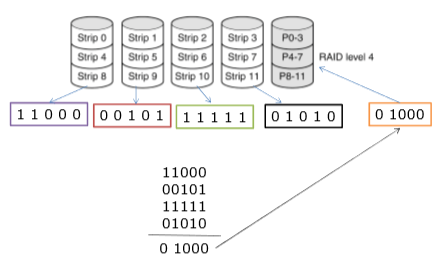
#### RAID 2

Questa istanza RAID utilizza striping unit grandi un bit e si compongono i disk a seconda della dimensione della parola (la quale determina anche la lunghezza del stripe). Ogni disco verrà utilizzato per contenere la ridondanza, essa viene calcolata attraverso i codici di Hamming (codici di correzione degli errori) i quali permettono la ricostruzione della sequenza originale in caso di errore. La lunghezza della ridondanza varia in base a quanti errori si vogliono correggere: più è lunga, più si riesce a ricostruire l’informazione, aggiungendo ulteriori dischi se necessario. Il costo effettivo equivale al numero di bit di ridondanza utilizzati mentre il tempo effettivo è dato dal disco più lento, dato che serve la sincronizzazione per un funzionamento corretto.

#### RAID 3

Il RAID 3 è molto simile al precedente con la differenza che si utilizza il bit di parità come ridondanza, le operazioni di lettura non necessitano di questa informazione a differenza di quella di scrittura che devono ricalcolarlo a ogni modifica delle informazioni. I bit di parità si memorizza solamente in uno dei dischi disponibili, tutti gli altri conterranno le informazioni effettive. Nel caso di rottura di uno dei dischi principali, vengono presi i bit di parità per ricalcolare il contenuto del disco rotto. Questa istanza risolve i problemi riguardanti lo spreco di dischi nel RAID 2, tuttavia tutti gli altri permangono e in più vi è un throughput basso, nonostante l’alta velocità.

#### RAID 4

Il funzionamento di questa istanza è analogo a quella del RAID 3 con la differenza che non più utilizzati dei bit ma dei blocchi come come striping unit. La striping unit di parità viene calcolata utilizzando l’operazione di XOR bit a bit con tutti gli altri blocchi della stripe, si utilizza la stessa operazione quando bisogna ricostruire i dati di un disco rotto. L’operazione di lettura coinvolge tutti i dischi perchè può avvenire in parallelo, per quanto riguarda la scrittura vi sono due possibili casi: se è richiesta la scrittura di un’intera stripe, tutto avviene simultaneamente, altrimenti, se bisogna solo scrivere alcune striping unit, bisogna anche ricalcolare la parità rileggendo quelle non modificate. Si può evitare di rileggere tutte le striping unit non modificate? Sì, basta utilizzare il blocco di parità vecchio per il calcolo di quello nuovo, utilizzando anche le striping unit che si intende modificare.

In questo modo si velocizza il cambio delle singole strip unit effettuando solamente 4 accessi al disco (2 in lettura e 2 in scrittura), questo problema è detto Small Write. L’unico problema di questa istanza RAID è il disco di parità stesso: essendo uno solo non è possibile effettuare scritture in parallelo per le striping unit di parità, facendo così da collo di bottiglia. Come si risolve questa situazione? Si divide la parità tra tutti i dischi del RAID.

#### RAID 5 - Parità distribuita

Il RAID 5 utilizza la tecnica Left-simmetric per distribuire la parità nei dischi, essa viene inserita nella posizione (n+i)%(n+1) del disk array (come si può vedere nell’immagine), tutto il resto rimane simile all’istanza precedente. La lettura di una singola striping unit rimane però un problema, oltre al fatto che può reggere solo la rottura di un disco.

#### RAID 6 - Doppia parità distribuita

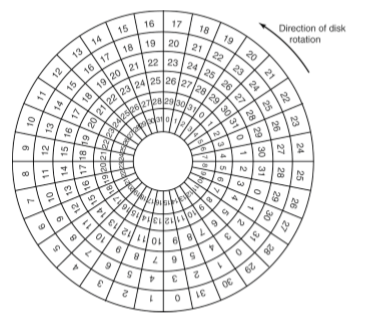
IL RAID 6 è un'istanza simile a quella precedente con la differenza che vi è un calcolo diverso per la ridondanza: i codici di Reed-Solomon. Questi codici generano due blocchi differenti per stripe, permettendo così di reggere la rottura di due dischi, entrambi possono essere calcolati uno in funzione dell’altro.

### Formattazione a basso livello del disco

### 

La formattazione a basso livello si occupa di definire la dimensione dei settori su ogni traccia del disco, essa definisce tre campi:

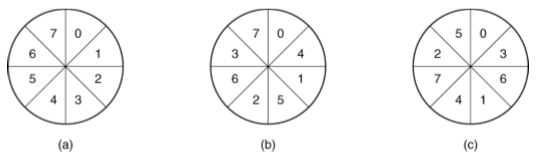
* Preambolo: parte in cui vengono scritte tutte le informazioni che servono al controller per gestire i settori, esso indica anche l’inizio di uno di essi;
* Data: Parte in cui vengono memorizzati i dati veri e propri del settore;
* ECC: parte riservata al codice di correzione degli errori per il rilevamento di essi nel campo Data, la lunghezza è data dal costruttore e più è grande, più è possibile correggere. Alla lettura di un settore, viene ricalcolato l’ECC e viene confrontato con quello presente nel postambolo del settore e, se è presente un errore, prova a correggerlo. Se la correzione dell’errore fallisce, esso viene segnalato al driver.

I driver possono vedere solamente il campo Data mentre il primo e l’ultimo sono visti esclusivamente dal controller. La capacità utile è data dal numero di settori che può memorizzare il disco per la quantità di memoria riservata al campo Data.

#### Skew

Per rendere più efficiente il disco si utilizza una tecnica detta Skew: essa consiste nel posizionare l’inizio dei settori appartenenti a tracce differenti un po’ più in là rispetto all’ultimo della traccia precedente, sfruttando in questo modo il tempo di spostamento della testina ed evitando così un ulteriore giro del disco. Il numero di settori da sfasare si trova dividendo il tempo di seek con il tempo di lettura d’un settore:

#### Skewing

Quando bisogna leggere i dati dati un settore, essi si memorizzano temporaneamente in un buffer del controller per poter calcolare l’ECC e, se non ci sono problemi, vengono trasferiti al driver. Il problema viene quando devo leggere due o più settore consecutivi: se eseguo il procedimento dettato prima per ogni singolo settore, devo aspettare un ulteriore giro perchè la traccia ha già passato quello successivo. Come fare in tal caso? si utilizza lo skewing, un meccanismo che permette di sfasare i settori consecutivi mettendoli più in là in modo da evitare ulteriori giri di disco. La distanza tra un settore e il suo successivo equivale al tempo di seek diviso il tempo di lettura di un settore

I buffer al giorno d’oggi possono contenere un’intera traccia e lì si fanno tutti i controlli del caso.

### Scheduling del braccio

Lo scheduling del braccio è utile per minimizzare il tempo di seek e, di conseguenza, massimizzare il throughput, per farlo si possono gestire opportunamente le richieste dei settori in base alla posizione della testina. Ogni richiesta pendente viene inserita in una coda e scelta utilizzando delle politiche di scheduling simili a quelle della CPU

### Politiche di scheduling

#### FCFS (First Come First Served)

FCFS è la più semplice delle politiche di scheduling e gestisce le richieste in base al loro ordine di arrivo. Questa soluzione però ha un tempo di seek troppo alto perché le richieste potrebbero interessare settori anche molto lontani tra loro.

#### SSTF (Shortest Seek Time First)

SSTF è una politica greedy che prende sempre le richieste più vicino alla posizione della testina. Questa politica è ottimale però causa starvation per le richieste troppo distanti e in più vi è l’unfairness, ovvero quando le richieste non vengono gestite con lo stesso tempo.

#### SCAN

SCAN è una politica che tiene traccia delle direzione della testina e delle richieste, gestendole in una sola direzione. Quando la testina arriva alla fine, essa va verso l’altra estremità dei settori. Questa politica permette di risolvere la starvation ma non l’unfairness, infatti potrebbe arrivare una richiesta da un settore già passato dalla testina.

#### C-SCAN

C-SCAN è una variante della SCAN che percorre circolarmente i settori, ritornando al punto di partenza ogni volta che arrivo alla fine. In questo modo risolvo l’unfairness.

#### 

#### 

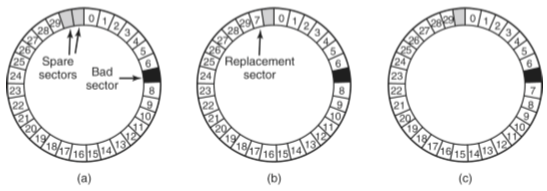
#### LOOK e C-LOOK

LOOK e C-LOOK sono varianti di SCAN e C-SCAN che tornano indietro quando arrivano all’ultima richiesta disponibile per quella direzione, tuttavia non si accorgono di eventuali richieste che arrivano dopo aver cambiato senso e quindi non torna indietro.

Esempio: mi trovo nel cilindro 11 e devo gestire le richieste 11, 1, 36, 16, 34, 9, 12 tra 0 e 39

| **Politica** | **Ordine** | **Seek** |
| --- | --- | --- |
| **FCFS** | 11, 1, 36, 16, 34, 9, 12 | 111 |
| **SSTF** | 11, 12, 9, 16, 1, 34, 36 | 61 |
| **SCAN** | 11, 12, 16, 34, 36, (39), 9, 1 | 66 |
| **C-SCAN** | 11, 12, 16, 34, 36, (39), (0), 1, 9 | 37 |
| **LOOK** | 11, 12, 16, 34, 36, 9, 1 | 60 |
| **C-LOOK** | 11, 12, 16, 34, 36, 1, 9 | 33 |

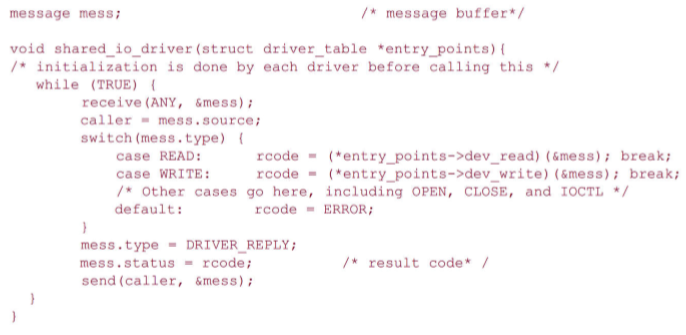
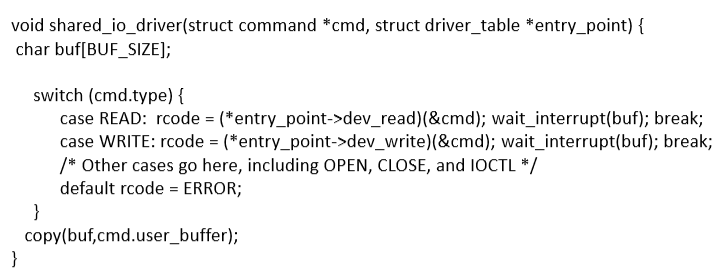
### Gestione degli errori del disco

La gestione degli errori fisici del disco può avvenire in due modi: attraverso il controller oppure utilizzando il driver. Nel primo caso, si utilizzano settori di scorta, detti sparetrack, invisibili al sistema operativo in cui il controller rimappa i settori danneggiati, memorizzando nel preambolo del precedente il relativo puntatore (caso b). Nel caso in cui la rimappatura fallisce, tutti i settori successivi vengono traslati di uno e si rimappa il settore danneggiato su quello immediatamente più avanti (caso c).

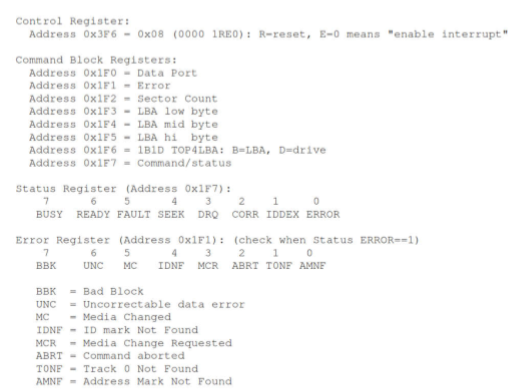
Nel caso in cui il controller non sia più capace di nascondere i settori, il sistema operativo esegue una scansione superficiale etichettando come non utilizzabili i settori danneggiati e quelli che lo saranno a breve, infine compila un file speciale in cui inserisce tutti i puntatori dei settori segnati. Vi sono anche gli errori dovuti allo spostamento del braccio, essi sono causati dall’allungamento termico del metallo e possono causare un movimento superiore alla norma o la lettura di un settore su una traccia differente da quella richiesta. In questi casi, l’errore viene notificato al drive il quale ricalibra il braccio, nel caso peggiore però potrebbe spegnere il disco per raffreddarlo.

### Meccanismi di interazione dei driver

L’interazione dei programmi utente coi driver avviene in maniera differente in base al tipo di kernel del sistema operativo. I kernel monolitici sono un insiemi di procedure eseguibili in modalità protetta dai processi utente, essi però hanno una difficile manutenzione. I microkernel risolvono il problema dei primi e funzionano fornendo solo i servizi essenziali, eseguibili in modo kernel, esso si occupa anche dei processi in modo utente e si interfaccia con le periferiche. La comunicazione tra i programmi utente e il kernel avviene tramite delle componenti attive che girano in modo utente. L’immagine a fianco mostra il funzionamento rispettivamento del microkernel e del kernel monolitico: Nel primo le frecce indicano i messaggi di richiesta e le risposte che si scambiano i processi, alcuni di essi hanno il compito di comunicare coi processi kernel quando necessario. A destra i processi utente accedono al kernel tramite le system call, permettendo così il passaggio a modo kernel. Nel modello microkernel i driver sono processi ed eseguono il seguente codice: Il codice effettua un’attesa attiva in cui si riceve il messaggio da inviare, si setta il mittente e si esegue una delle funzioni che sono puntate da dev\_read e dev\_write in base al tipo (se non è specificato, viene restituito un errore. Alla fine si setta lo stato del messaggio e si invia.

Nel sistemi con kernel monolitico, invece, i driver sono parti di codice del kernel stesso, eseguibili dai processi utente:

La scelta della funzione da chiamare avviene in modo analogo al codice del microkernel, l’unica differenza è alla fine di esse il processo chiamante rilascia la CPU e viene messo in attesa in una coda.



### 

I driver utilizzano un’interfaccia, detta IDE, contenente dei registri, essa permette ai driver di interfacciarsi con le periferiche. Il control register è formato da 8 bit in cui i primi 4 sono a 0, gli ultimi invece sono 1RE0, dove R è il reset se uguale a 1 mentre E è l’enable interrupt se uguale a 0.

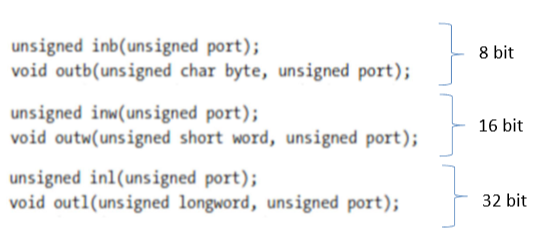
Gli 8 command block register svolgono diverse funzioni, ognuna assegnata a un registro:

* Data Port: permette di leggere/scrivere dati su un settore;
* Error: segnala la presenza di errori;
* Sector Count: indica quanti settori consecutivi si vogliono leggere o spostare nell’indirizzo dato dai 4 registri che seguono;
* LBA low byte: primi 8 bit del Local Byte Address;
* LBA mid byte: 8 bit centrali del Local Byte Address;
* LBA high byte: successivi 8 bit del Local Byte Address;
* 1B1D TOP4LBA: bit più significativi di LBA in cui B indica se è LBA o no mentre D indica il drive (0 se è master e 1 se è slave);
* Command/status: manda il comando al driver e si sospende, quando vi è un interrupt, sceglie il driver che ha eseguito l’operazione.

Lo status register è grande 8 bit e indica lo stato del registro attraverso delle costanti:

* BUSY: indica che il settore è pieno;
* READY: indica che il settore è pronto a essere letto/scritto;
* FAULT: indica che un settore è danneggiato;
* SEEK: indica che vi è un seek in esecuzione;
* DRQ;
* CORR;
* IDDEX;
* ERROR: Indica la presenza di un errore, esso è contenuto nell’Error Register quando presente.

Ultimo ma non meno importante, l’Error Register contiene l’errore quando lo status register ha ERROR=1:

* BBK (Bad block): indica che il blocco che si vuole utilizzare non è disponibile alla lettura/scrittura perché è danneggiato o corrotto;
* UNC (Uncorrectable data error): indica un errore persistente;
* MC (Media Changed): indica che il dispositivo potrebbe essere stato cambiato;
* IDNF (ID mark not found);
* MCR (Media Change Requested);
* ABRT (Command Aborted);
* T0NF (Track 0 not found);
* AMNF (Address mark not found).

La lettura/scrittura nei registri avviene attraverso funzioni di libreria in kernel, la differenza sta nella parola che bisogna leggere/scrivere. Esistono anche versioni di queste funzioni che permettono la lettura/scrittura di sequenze di parole.

#### 

#### 

#### 

#### 

#### Funzione ideinit()

| void ideinit()  {  int i;  initlock(&idelock,”ide”);  picenable(IRQ\_IDE);  iopicenable(IRQ\_IDE,ncpu=1);  idewait(0);  /\*Controlla se il disco 1  è presente \*/  outb(0x1f6,0xe0 | (1<<4)); [1]  for(i 00;i<1000;i++)  if(inb(0x1f7)!=0) [2]  {  havedisk1=1;  break;  }  // ritorna al disco 0  outb(0x1f6,0xe0 | (0<<4)); [3]  } |
| --- |

picenable(IRQ\_IDE) e iopicenable(IRQ\_IDE,ncpu=1) sono funzioni di libreria che permettono di abilitare l’I/O e l’IDE.

1- Nel registro 0x1f6 scrivo l’indirizzo 0xE0 (equivalente a 0b11100000) in OR con 0x1 shiftato di 4 bit, il risultato è 0b111**1**0000,il bit D=1 (quello segnato in grassetto), indica il disco 1.

2- si controlla il contenuto dello status register (corrispondente a 0x1f7) per verificare se il disco 1 è presente, infatti se il contenuto del registro è uguale a 0, significa che si sta riferendo al disco 0 ( il cui codice è 0b11100000). Se il contenuto dello status register è diverso da 0, significa che il disco 0 è presente e quindi si esce dal ciclo. Il numero 1000 è stato dato per dare al disco 1 il tempo di connettersi.

3- Alla fine ritorno al disco 0 facendo la medesima operazione del punto 1, l’unica differenza è che utilizzo 0x0 al posto di 0x1.

#### Funzione idewait(int checkerr)

| static int idewait(int checkerr)  {  int r;  while(((r=inb(0x1f7) & (IDE\_BSY|IDE\_DRDY))!=IDE\_DRDY); [1]  if(checkerr && (r & (IDE\_DF|IDE\_ERR))!=0) return -1; [2]  else return 0;  } |
| --- |

IDE\_BSY (0x80) e IDE\_DRDY (0x40) sono costanti che indicano rispettivamente se il disco pieno o se è ready, IDE\_DF (0x20) e IDE\_ERR (0x01) invece indicano dei codici errore.

1-se il disco è busy, estraggo il valore dallo status register e glielo inserisco.

2- quando checkerr è uguale a 1, si abilita il controllo degli errori: se il contenuto di r non corrisponde né a IDE\_DF né a IDE\_ERR, viene restituito 0, ovvero non vi sono stati errori, altrimenti restituisce -1.

Per verificare se il disco è busy/ready e il controllo degli errori, viene utilizzato un AND bit a bit a bit (&).

#### 

#### Struttura del buffer

| struct buf  {  int flags; //campo che contiene valori che indicano il tipo di operazione, esse sono codificate nei tre define in basso  uint dev; //indentifica il settore del disco su cui leggere/scrivere (MINOR number)  uint blockno; //indentifica il disco (MAJOR number)  struct buf \*prev; //puntatore al blocco precedente  struct buf \*next;//puntatore al blocco successivo  //sono utilizzati per la LRU cache list  struct buf \*qnext;//richieste pendenti del driver  uchar data[BSIZE];//dati da leggere/scrivere  }  //OPERAZIONI  #define B\_BUSY 0x1 //buffer bloccato  #define B\_VALID 0x2 // buffer appena letto dal disco  #define B\_DIRTY 0x4//il buffer ha bisogno di una scrittura su disco |
| --- |

#### 

#### Funzione idestart (struct buf \*b)

| #define IDE\_CMD\_READ 0x20 //permette la lettura di un blocco  #define IDE\_CMD\_WRITE 0x30 //permette la scrittura di un blocco  #define IDE\_CMD\_RDMUL 0xc4 //permette la lettura di una sequenza di blocchi  #define IDE\_CMD\_WRMUL 0xc5 //permette la scrittura di una sequenza di blocchi  static void idestart(struct buf \*b)  {  if(b==0) panic(“idestart”);  if(b->blockno>=FSSIZE) panic(“incorrect blockno”);  int sectorPerBlock=BSIZE/SECTOR\_SIZE;  int sector=b->blockno\*sectorPerBlock;  int readCmd=(sectorPerBlock==1) ? IDE\_CMD\_READ : IDE\_CMD\_RDMUL;  int writeCmd=(sectorPerBlock==1) ? IDE\_CMD\_WRITE : IDE\_CMD\_WRMUL;  if(secotrPerBlock>7) panic (“idestart”);  idewait(0);  outb(0x3f6,0); //genera interrupt  outb(0x1f2,sectorPerBlock); //numero di settori  outb(0x1f3, sector&0xff);  outb(0x1f4, (sector>>8)&0xff);  outb(0x1f5, (sector>>16)&0xff);  outb(0x1f6, 0xe0|((b->dev&1)<<4) | ((sector>>24)&0x0f));  if(b->flags &B\_DIRTY)  {  outb(0x1f7, writeCmd);  outsl(0x1f0,b->data,BSIZE/4);  }  else outb(0x1f7,readCmd);  } |
| --- |

All’inizio si controlla che l’input sia corretto con una serie di controlli, è bene farlo onde evitare crash di sistema. panic fa terminare il driver inserendo un messaggio di errore. BSIZE/SECTOR\_SIZE indicano quanti settori sono presenti in ogni blocco, questo valore permette di scegliere tra la lettura/scrittura singola o multipla, utilizzando l’operatore ternario. idewait(0) esegue un'attesa attiva fino a quando il disco non è ready, in seguito :

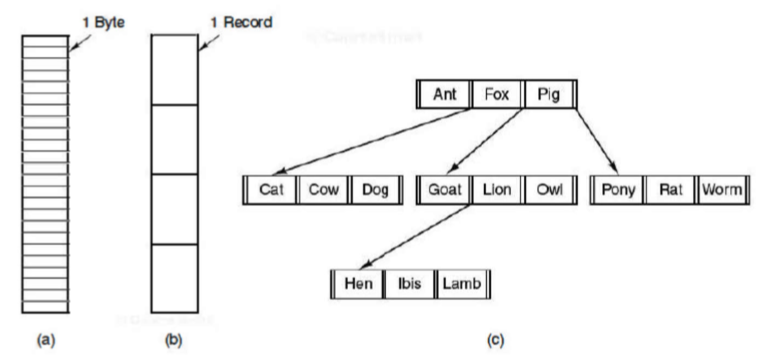
* si abilitano le interruzioni scrivendo 8 nella porta I/O 0x3f6, in realtà il valore 8 non cambia il bit E del registro, sul binario, infatti, non lo setta;
* Scrivo quanti settori sono coinvolti nella porta 0x1f2;
* Scrivo l’indirizzo del primo settore (che a volte è anche l’unico), dato che gli indirizzi LBA sono da 28 bit, lo si “spezza” in 4 parti shiftando di 8, 16 e 24 bit a destra e scrivendo il risultato rispettivamente nelle porte 0x1f3, 0x1f4, 0x1f5 e 0x1f6 utilizzando delle maschere di bit. Gli 8 bit meno significativi, quelli centrali e i successivi vengono trovati con la maschera 0xff, corrispondente a una serie di 1, e facendo l’AND bit a bit con esso. I 4 bit più significativi, invece, shifta d 24 bit e utilizza una maschera diversa, ovvero 0x0f ( corrispondente a 0b00001111). In più bisogna tenere conto del campo dev del buffer e lo si mette in AND bit a bit con ox1 per verificarne la parità: se è pari, il dispositivo è il master del controller. altrimenti è lo slave. Il risultato ottenuto deve poi essere shiftato di 4 bit a sinistra e messo in OR bit a bit con la parte trovata in precedenza;
* Si utilizza il campo flag per capire quale operazione è stata eseguita: se il blocco è dirty, si inserisce non command register il comando di scrittura e si utilizza outsl (32 bit) per scrivere i dati nella porta dati in una certa dimensione (BSIZE/4). Se invece il blocco è clean, scrivo nel control register l’operazione di lettura e l’hardware fa tutto il resto.

### File System

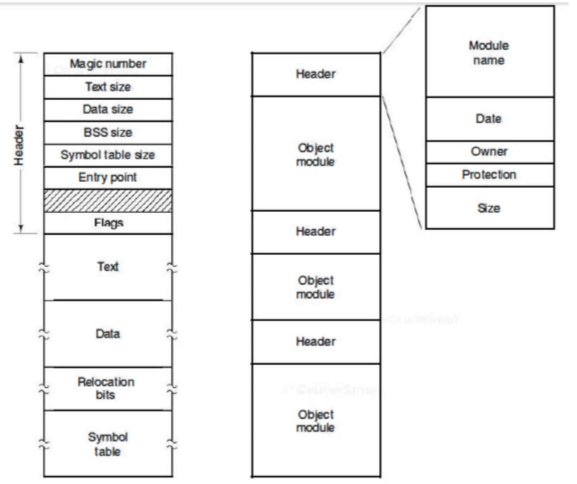
Il file system è un’astrazione del sistema operativo formata da strutture dati e meccanismi che la implementano, essa permette la gestione di file e cartelle in modo affidabile ed efficiente. Dal suo punto di vista, l’utente può vedere solamente le astrazioni del file system e può eseguire operazione su di esse, il sistema operativo implementa il tutto adeguatamente affinché funzioni in modo corretto.

#### Naming e struttura dei file

I file devono avere nomi legittimi che hanno un significato per l’utente, essi possono contenere solamente caratteri relativi allo standard ASCII e Unicode. Al giorno d’oggi, un file può essere strutturato nei tre seguenti modi:

* Flusso di byte non strutturato (a): questa struttura permette di ottenere la massima flessibilità, tuttavia vi è poco supporto da parte del sistema operativo, viene tutt’oggi utilizzato per Windows e UNIX;
* Flusso di record (b): i file sono strutturati utilizzando una sequenza di record di lunghezza fissa, rispetto al caso precedente vi è un supporto maggiore al costo di una minore flessibilità, tutt’oggi non viene più utilizzato per nessun tipo di problema;
* Albero di record (c): questa struttura è molto simile al flusso di record con la differenza che è organizzato ad albero, l’accesso ai file è più diretto e viene utilizzato del campo dei Big Data.

#### Tipi di file

I file possono essere di tipi diversi a seconda del ruola che hanno, solitamente però sono divisi in file regolari e file speciali. I file regolari sono quelli in cui è contenuta l’informazione utente, essi possono essere di testo, binari, oggetto o eseguibili, quelli speciali possono contenere ulteriori file o identificare i device. macOS, a differenza di Windows e Linux, utilizza la Resource Fork, ovvero un sistema di tipizzazione che assegna a ogni file un tipo, esso non può essere cambiato se non con operazioni opportune. A conferma di ciò, se si prova a cambiare l’estensione di un file, Window e Linux cambiano anche l’applicazione con cui aprirlo mentre macOS utilizza sempre lo stesso. Per distinguere i file si utilizza un campo specifico detto Magic Number, esso infatti permette di riconoscere il tipo di file in modo da evitare errori.

#### Accesso ai file

L’accesso a un file può avvenire in modo sequenziale o casuale. Col primo metodo si accede ai blocchi del file sequenzialmente, di conseguenza bisogna leggere tutti i blocchi precedenti prima di quello scelto, questa lettura però permette di riavvolgere il file quando necessario. Nel modo casuale, invece, vi è un accesso diretto al blocco scelto o comunque posso evitare di leggere tutti quelli che lo precedono.

#### Attributi di un file

Gli attributi sono voci extra del file file che permettono di descriverne i dati quali il tipo di protezione, la password (se presente) e altre informazioni riguardanti i flag e la data. Nella tabella a fianco vi sono alcuni tipi di attributi di un file.

#### Operazioni sui file

Un file è paragonabile a un oggetto avente i suoi attributi e i suoi metodi, su esso è possibile eseguire:

* Create: Viene creato un file vuoto e vengono definiti i suoi attributi;
* Delete: Cancellazione di un file;
* Open: Viene aperto un file, portandolo in RAM insieme ai suoi attributi per velocizzarne l’accesso o la modifica, e viene segnalo che lo è sulla tabella dei file aperti del file system.
* Close: si aggiornano le strutture dati del file e lo si toglie dalla RAM, avvisando anche il file system il quale lo toglierà dalla tabella dei file aperti;
* Read/Write/Append/Seek: Operazioni di lettura o modifica del file;
* Get/Set attribute: permettere di leggere o scrivere l’attributo di un file;
* Rename: Permette di rinominare il file;

#### Operazioni su directory

Le operazione delle directory sono diverse da quelle dei file per motivi di efficienza, esse sono:

* Create/Delete Dir: creazione/cancellazione di una directory inizializzando/eliminando le strutture dati che la caratterizzano;
* Open/Close DIr: apertura/chiusura di una directory;
* Read Dir: elenca il contenuto della directory;
* Link/Unlink: crea/cancella un riferimento a un file in una cartella.

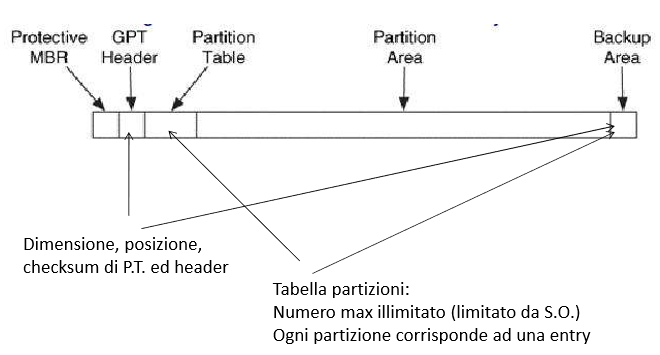
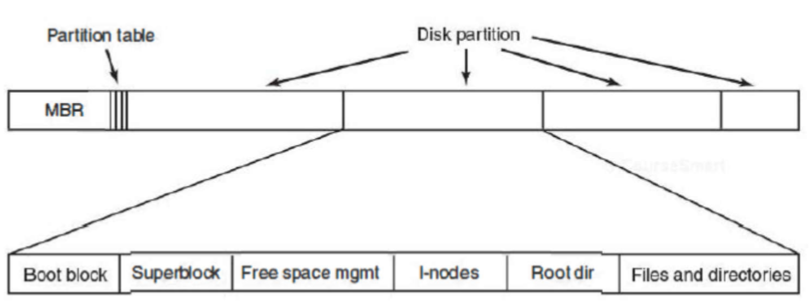
### Organizzazione logica di un file system

Un file system è organizzato in modo gerarchico e ad albero in cui le foglie rappresentano i file e i nodi le cartelle. File e cartelle vengono rappresentate attraverso il pathname, ovvero l’insieme di cartelle che bisogna percorrere per arrivare al file di destinazione, esso può essere assoluto (quando parte dalla root) o relativo (quando inizia dalla directory corrente)

### Implementazione di un file system

L’implementazione di un file system è una struttura dati allocata in una partizione di memoria del disco insieme ai dati. una partizione è una sequenza di settori indipendente da tutte le altre, tutte le informazioni relative a esse sono contenute nella partition table. I sistemi di partizione sono MBR e GPT.

#### MBR (Master Boot Record)

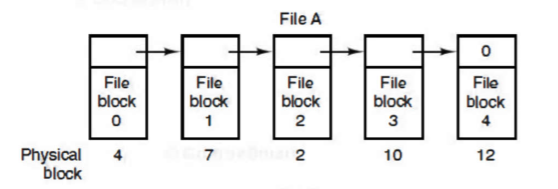
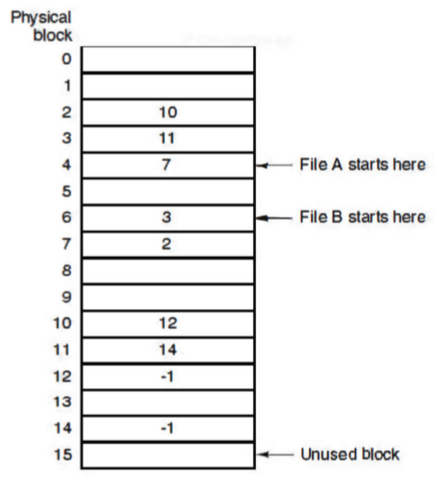
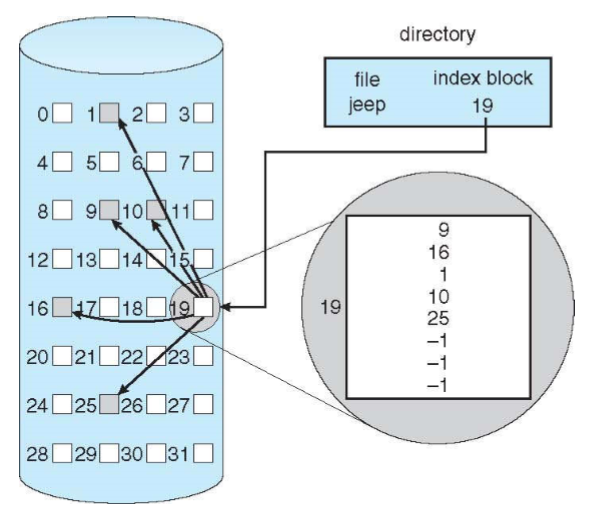
Questo sistema presenta il primo settore come riservato a MBR e alla partition table ed è strettamente correlato col BIOS perchè deve sapere qual è il sistema operativo da far partire, la prima cosa che fa infatti è leggere il bootblock, un blocco che contine il codice in grado di avviare il sistema operativo per localizzare la partizione attiva e quindi caricare il sistema operativo. Localizzata la partizione, si va a leggere il superblock, lì ci sono tutti i parametri chiave relativi al file system. Altri blocchi della partizione sono il free space management (indica quali sono i cluster liberi e quali no), la parte delle i-node (strutture che gestiscono le proprietà dei file), la root e la parte dedicata a file e cartelle.Questo sistema permette di supportare al massimo 4 partizioni, anche se vi sono meccanismi che permettono di aumentarne il numero. Tuttavia vi è un problema: se il primo settore si danneggia, perdo tutte le informazioni riguardanti le partizioni o comunque potrei perderne l’accesso.

#### GPT (GUID Partition Table)

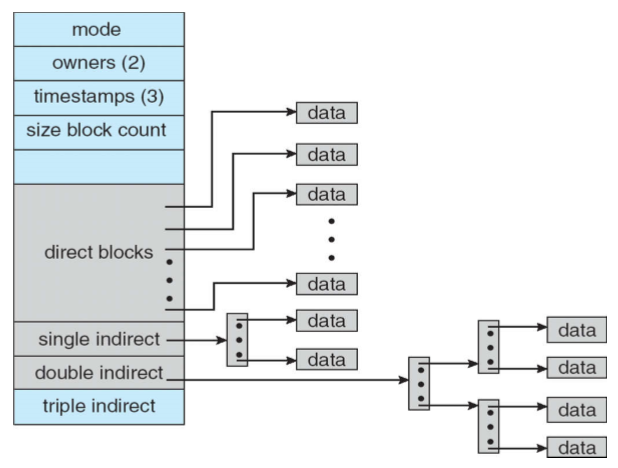
In questo sistema il primo settore è il protective MBR, utilizzato per la retrocompabilità con le vecchie versioni del BIOS) mentre il secondo è il GPT Header, in essa si trovano le informazioni relative alla partition table e un checksum alla fine di essa. Dopo di essa vi è la partition table, essa può contenere tante partizioni quante ne può contenere il disco (a livello logico sono infinite), In tutti gli altri settori del disco vi è un backup del GPT Header.

#### Implementazione dei file

Un file può essere implementato nei seguenti modi:

* Allocazione Contigua: assegno al file dei blocchi appartenenti alla stessa traccia in modo da minimizzare i tempi di seek e gestendo in modo efficiente lo spazio. Questa tecnica però crea frammentazione quando viene cancellato un file (risolvibile con la deframmentazione del disco) e in più non permette una corrette gestione delle modifiche ai file;
* Lista linkata di blocchi: Viene salvata la testa della lista nel file system, tutti gli altri nodi sono connessi uno dopo l’altro tramite dei puntatori. Nonostante risolva i problemi del punto precedente, questa implementazione è rischiosa dato che nel caso si dovesse danneggiare un nodo, non riuscirei più a trovare quelli successivi a esso;
* FAT (FIle Allocation Table): La FAT è un particolare tipo di lista in cui si utilizza un array per memorizzare i blocchi, ogni cella conterrà anche l’indirizzo alla prossima cella.La FAT però potrebbe essere più grande della RAM stessa e, in più, si inserisce in memoria file che in quel momento non sto utilizzando;
* Allocazione Indicizzata: nel file system salvo il primo blocco e carico tutti gli altri in memoria quando ce n’è bisogno.

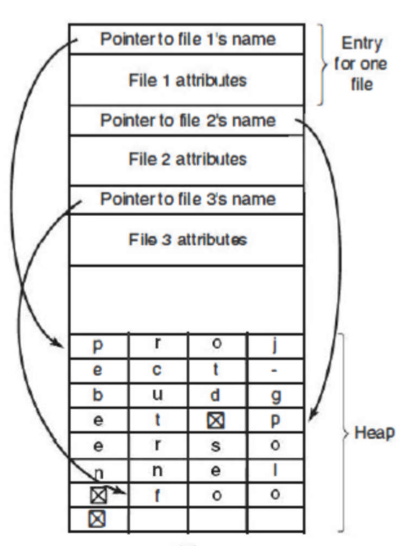
#### Dimensioni del blocco indice

La dimensione del blocco indice è importante perché se troppo piccolo limita la grandezza del file mentre se troppo grande causa frammentazione, In più vi è un ulteriore problema: se un settore si danneggia, non posso più indicizzare ai blocchi indice successivi a esso. Per risolvere questo problema, si utilizzano gli indici multilivello, una soluzione flessibile che permette di espandere/ridurre la grandezza di un file utilizzando due tipi di blocchi:

* Outer Index: blocchi che indicizzano Index Table;
* Index Table: blocchi che indicizzano i blocchi di memoria dei file.

E’ possibile ottimizzare lo spazio inserendo alcuni blocchi dei file direttamente alla i-esima allocazione

### Implementazione delle cartelle

L’implementazione delle cartelle si trova nella stessa parte in cui è presente quella dei file, essa è formata da righe, dette directory entry, i quali possono indicare file o ulteriori cartelle. Gli attributi di un file vengono memorizzati nella directory entry relativa al file stesso, tuttavia, se non vi è più spazio, si indicizzano utilizzando un’apposita struttura dati. I nomi dei file e delle cartelle sono a loro volta contenuti nella directory entry, essi però non devono essere di lunghezza fissa però altrimenti causerebbero frammentazione, di conseguenza la parte dedicata al nome è variabile e pesca i caratteri da un pool. La soluzione precedente, però, causa ancora frammentazione, di conseguenza si riferisce la parte variabile in un pool di caratteri contenente i nomi.

#### Ricerca efficiente di un file in una cartella:

* Una possibile soluzione è organizzare le directory entry secondo determinati criteri e scorrerle tutte, essa però va bene nelle directory entry piccole e non in quelle grandi perché la complessità è lineare;
* Si può utilizzare una tabella hash per rappresentare le tabelle in cui, dato il nome di un file, lo si converte nel valore del bucket. Questa soluzione permette di ottenere liste corte e omogenee ( di conseguenza, ci metto meno tempo a trovare il file), tuttavia vi è inefficienza nell’inserimento e nella cancellazione (bisogna fare due operazioni per ogni oggetto);
* Si possono combinare le due precedenti soluzioni impostando una soglia: se la grandezza del file è minore, utilizzo la lista linkata, altrimenti la cambio con la tabella hash.

### File Condivisi

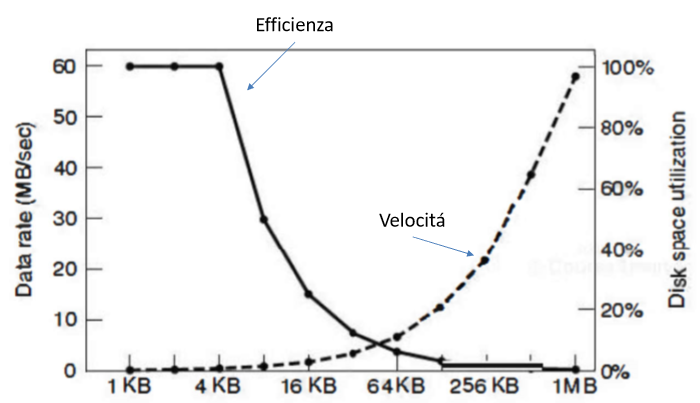
I file condivisi permettono l’accesso a essi da parte di più utenti o più risorse, essi si possono implementare creando due directory entry che riferiscono allo stesso file. Dentro le directory entry ci vanno i cluster del file, tuttavia questo non permettere a un utente di vedere le modifiche fatte da altri, di conseguenza si inserisce una struttura dati simile alla i-node che memorizza gli indirizzi dei cluster (Hard link), risolvendo così il problema. In questa struttura dati è anche presente un contatore il quale segna il numero di cartelle a cui il file referenzia. Quando un utente cancella un file condiviso, viene decrementato il contatore nella i-node anche se a cancellarlo è il proprietario, questo è un problema perché si spreca memoria inutilmente. Questo problema si può risolvere utilizzando i link simbolici in cui un utente possiede il file originale mentre tutti gli altri hanno un ulteriore file contenente il pathname del primo (Soft link). In questo modo, alla cancellazione del file originale, non bisogna andare a modificare tutti i collegamenti, tuttavia è inefficiente a livello di accesso al file.

### Virtual File System (VFS)

Nel caso abbia più file system in una periferica, è possibile accedere a tutti simultaneamente? sì e per farlo si utilizza il Virtual FIle System, esso è un insieme di astrazione che si interpone tra i processi e i file, fornendo un’unica interfaccia. L’interfacciamento avviene tramite le system call, il VFS traduce la chiamata di sistema con quella adeguata al file system in cui si trova il file richiesto. Nel VFS è presente un superblock per ogni file system montato, esso fornisce informazioni relative al file system stesso e al suo layout, vengono memorizzati in una struttura dati apposita. La v-node è una struttura che permette di riferire le i-node (ognuna descrive uno e un solo file) di un file indipendentemente dall’implementazione di quest’ultimo. La directory entry rappresenta la voce di una directory, queste ultime sono contenute in una cache apposita. Al collegamento di un file system, esso viene registrato dal VFS passando gli indirizzi delle system call e memorizzandole insieme ad altre informazioni in un superblock. Quando bisogna aprire un file, si va a prendere la root directory dal superblock (tutte le altre cartelle verranno prese di conseguenza). Una volta trovato il file inserisco la sua i-node e altre informazioni in una v-node, quest’ultima verrà poi inserita in una tabella apposita. Un volta aperto il file, tutte le operazione verranno svolte sul file descriptor.

### Gestione dello spazio del disco

#### Dimensione dei blocchi

La gestione dello spazio del disco è importante perché permette di rendere più efficiente le operazioni, tra questi vi è anche il problema riguardante la dimensione dei blocchi, infatti si hanno troppi tempi di seek se troppo piccoli e frammentazione se troppo grandi, si è così deciso di utilizzare una dimensione media in modo da avere entrambi i vantaggi. Nella tabella a fianco vi è la lunghezza del file è tre percentuali che riferiscono a tre differenti periodi storici, esse indicano la distribuzione delle dimensioni dei file: con blocchi da 1KB, c’è una probabilità tra il 30% e il 50% che il file stia in uno di essi. All’aumentare delle dimensioni del blocco diminuisce l’efficienza, qual è la scelta migliore? Si privilegia una delle due in base alle esigenze, storicamente però si preferiscono cluster piccoli per avere la massima efficienza. Questo problema non si pone nelle memorie a stato solido dato che le velocità sono simili per ogni dimensione del blocco. La velocità di lettura di un settore di k byte si calcola nel seguente modo:

#### Gestione dello spazio libero

Oltre alla dimensione dei blocchi, c’è anche bisogno di gestire lo spazio libero, lo si fa utilizzando un’apposita struttura dati in grado di tenere traccia dei blocchi liberi. Una prima soluzione è l’utilizzo di una lista linkata nel free space management, contenente l’indirizzo dei blocchi liberi, il problema è che l’allocazione è scomoda per i file contigui. Per risolvere questo problema, è possibile ordinare la lista, tuttavia le operazione su di essa diventano inefficienti. in più, la dimensione della suddetta lista è grossa dato che cresce all’aumentare dello spazio di indirizzi, quindi vi è anche uno spreco di spazio. Per questo motivo si utilizza una bitmap grande quanto il numero di blocchi utilizzati in cui ogni bit corrisponde allo stato di un cluster (1 se è occupato, 0 se è libero).

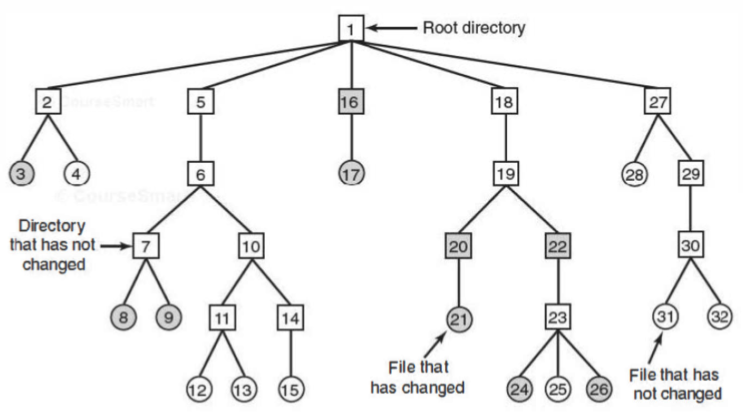
Esempio: calcolo delle dimensioni della lista e della bitmap:

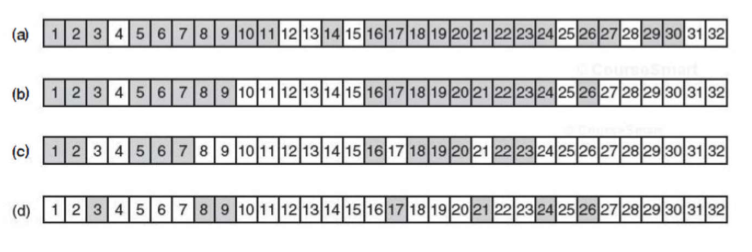
DimBlocco= 1KB DimIndirizzi: 32 bit Ddati= 512 GB

| LISTA  #Ind=1K/(32/8)=1024/4=256  #Blocchi=512G/1K=512M  #nodi=#blocchi/#ind=512M/256=2M  DimLista=#nodi\*DimBlocco=2M\*1K= 2GB | BITMAP  #Ind=1K/(32/8)=1024/4=256  #Blocchi=512G/1K=512M  DimBitmap=#Blocchi/8=512M/8=64 MB |
| --- | --- |

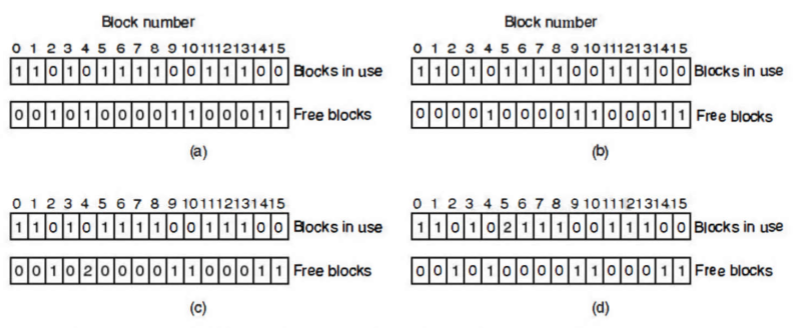
### Affidabilità di un file system

L’affidabilità di un file system è garantita attraverso i backup, ovvero delle copie di dati su partizioni esterne. Il backup però ha i seguenti svantaggi: si spreca spazio per dati ridondanti e ci vuole molto tempo per farlo, una possibile soluzione è escludere file speciali ed eseguibile che, oltre a risparmiare memoria, permettono di evitare situazioni dannose. I backup possono essere di due tipi: quelli incrementali permettono solamente il salvataggio dei file con modifiche recenti mentre quelli completi, come dice il nome, salvano tutti i file. Esistono diverse tecniche per effettuare un backup:

* Tecnica di base: vi è una baseline, un backup sul quale si basano quelli incrementali, su cui effettuo ulteriori backup incrementali in cui inserisco tutti i file modificati, c’è però un problema: vi è ridondanza tra i backup incrementali, un backup infatti contiene tutti gli elementi salvati in quello precedente;
* Base Ottimizzato:questa tecnica è una versione ottimizzata di quella precedente in cui i backup incrementali salvano i file modificati senza inserire quelli del precedente, c’è però un problema: nonostante lo spazio sia minore, ho bisogno di tutti i backup precedenti per effettuare un ripristino;
* Dump fisico: in questa tecnica vengono letti i cluster dal primo all’ultimo dell’area utente senza utilizzare le strutture dati del file system, è veloce e semplice da utilizzare (da parte del backup) ma è complesso e non c’è modo di evitare i cluster non allocati, sprecando di conseguenza spazio.
* Dump fisico ibrido: versione della tecnica precedente in cui vengono letti i cluster facendo riferimento alla bitmap per rilevarli, tuttavia, oltre a perderle la sequenzialità dei cluster, i settori danneggiati rallenterebbero il backup stesso, dato che sono segnati come occupati;
* Dump Logico: questa tecnica memorizza file e cartelle utilizzando le astrazioni del file system, essa esegue 4 passate sui file/ cartelle nel seguente modi:
  + vengono mercati tutti i file modificati e tutte le cartelle;
  + si smarcano tutte le cartelle che non contengono elementi modificati, sia diretti che in ulteriori sottocartelle;
  + tutte le cartelle marcate vengono memorizzate nel backup e smarcate;
  + tutti i file vengono memorizzati nel backup e smarcati.

Vi è però un problema: dato che non memorizzo le strutture dati relative al file system, occorre ricostruire la free list, per farlo vi sono due metodi: o si tiene traccia dei cluster utilizzati durante il restore, facendo il completamento della vecchia free list, oppure si cancellano i cluster da essa ogni volta che ne vengono allocati. Un altro problema riguarda i link ai file perchè, dato che la posizione del file viene persa al momento del backup, non è possibile ricostruirli. In più vi sono i file sparsi in cui la dimensione logica è diversa da quella fisica, quindi bisogna agire di conseguenza.

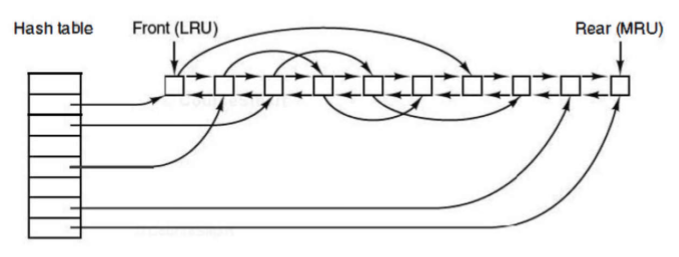
### Controllo della consistenza

Il controllo della concorrenza viene effettuato attraverso un programma detto FSCK (File System ChecK), esso utilizza due array di interi i quali rappresentano quante volte un cluster è allocato e quante compare nella free list, nelle situazioni normali i due array sono complementari (a). Quando il sistema operativo riparte, legge la free list e in seguito legge le i-node di ogni elemento. aggiungendo 1 alla posizione corrispondente. Nel caso ci sia un cluster nè libero nè allocato (b), si può trattare di un file cancellato aggiornato in memoria ma non nella free list a causa di un crash del sistema, si risolve mettendo a 1 la rispettiva cella della free list. E’ anche possibile che un blocco sia duplicato all’interno della free list © e si verifica quando si aggiunge un blocco ad essa e va in crash prima di trovare i duplicati, in tal caso si ricostruisce la lista dei blocchi liberi. L’ultimo caso di inconsistenza è presente quando un blocco duplicato tra quelli utilizzati, si verifica quando il sistema va in crash nel momento in cui la i-node di un file viene rilascia e riassegnata a un altro senza riuscire ad aggiornarli sul disco. Per risolvere questo problema si assegna a uno di due file un’altro blocco e si chiede all’utente quale dei due è quello giusto, agendo di conseguenza. 

### Eliminazione delle inconsistenze

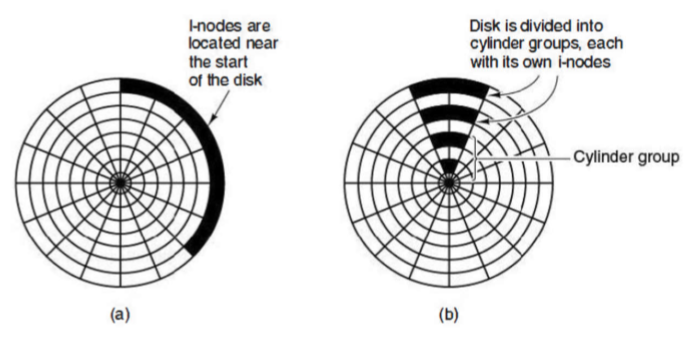
L’eliminazione delle inconsistenze è possibile grazie ai journalist file system, le cui operazioni vengono eseguite in modo atomico e si scrivono in un journal, un file speciale gestito dal file system stesso e invisibile agli utenti. Quando le operazione del log vengono completate, esse vengono tolte dal file, in questo modo il sistema riesce a capire se c’è stato un crash o meno nel caso in cui il journal sia vuoto o meno, si ripristina lo stato precedente e si rieseguono le operazioni da capo nel caso in cui il journal sia pieno. Le operazioni, per essere rieseguibili, devo essere idempotenti, ovvero il loro risultato non deve cambiare lo stato precedente alla seconda esecuzione di esse.

### Efficienza del file system

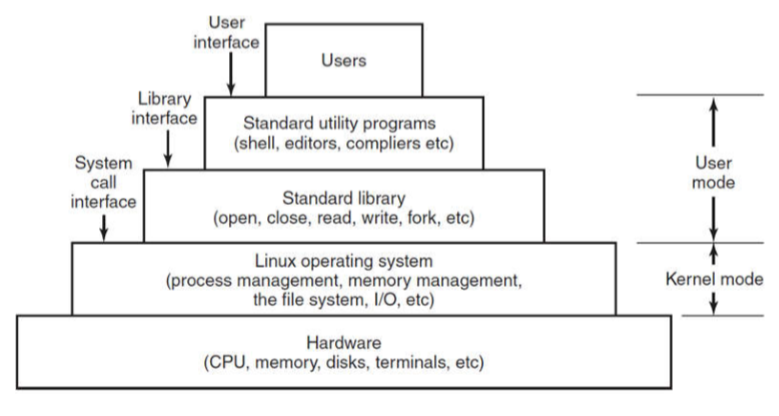
Per rendere efficiente un file system, si può implementare una cache in cui memorizzare i blocchi, essa funzionerà come una tabella hash. Nel caso non abbia più spazio per inserire ulteriori blocchi, utilizzo l’algoritmo di rimpiazzamento LRU che rimpiazzerà quello meno usato recentemente, esso è oneroso da implementare però lo si preferisce agli altri perché non accedo ai blocchi tanto frequentemente quanto le pagine. LRU si implementa con una lista doppiamente linkata, di conseguenza ogni blocco apparterrà al massimo a due liste: la prima è quella della rispettiva cella della tabella hash mentre la seconda è LRU stessa. Questa politica funziona quando LRU lavora con dati non critici, se così non fosse, essi sarebbero da finalizzare sul disco il prima possibile quando vengono modificati, è comunque possibile scansionare a intervalli regolari la cache per scrivere su disco tutti i cluster modificati senza toglierli. 

Un altra tecnica è il pre-fetching: si caricano in cache blocchi prima che mi servano, utilizzando le proprietà di località:

* Tempo: si richiedono i blocchi di un file successivi a quelli scelti;
* Spazio: si richiedono i blocchi di un file adiacenti a quelli scelti.

In questo modo però corro il rischio di eliminare cluster utilizzati da altri con uno che, per il momento, non verrà utilizzato, per evitare ciò si eseguono analisi euristiche in cui, se l’utente sta leggendo sequenzialmente il file, probabilmente leggerà anche quelli seguenti. La lettura in modo casuale però non permette di capire quali blocchi scegliere, per questo motivo si è deciso di inserire un bit di lettura (1 se sequenziale e 0 se casuale) in cui il prefetching si adatterà in base alla situazione. L’ultima tecnica è legata alla struttura delle partizioni: alla lettura di un file, si porta in memoria la relativa i-node per recuperare i blocchi direttamente da essa, se è troppo grande, però, oltre a perdere tempo a leggere gli indirizzi nella i-node, devo anche prendere i blocchi nella parte dati. Una soluzione per questo problema è disporre i cluster di un file insieme nella relativa i-node, in questo modo si può sapere in quale parte del disco si trovano i blocchi indirizzati dalla i-node stessa. 

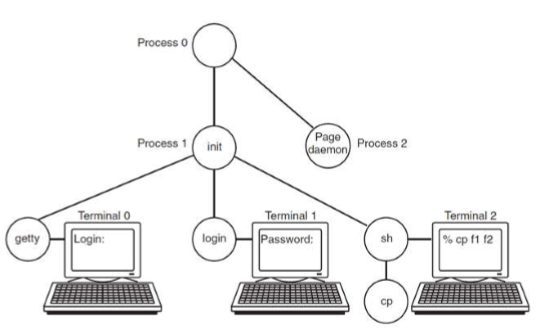
### Linux

Linux è un sistema operativo progettato per gestire molteplici process in contemporanea, esso comprende funzionalità che consentono agli utenti di lavorare e condividere dati in maniera controllata. Un sistema Linux è implementato come una piramide. Alla base di essa c’è l’hardware, il quale comprende tutte le risorse del calcolatore, al di sopra vi è il sistema operativo il cui compito è fornire un'interfaccia di system call ai programmi dei livelli successivi, la cosiddetta libreria standard. Prima degli utenti veri e propri, vi sono i programmi standard, una serie di programmi utili all’utente stesso per poter utilizzare il calcolatore. 

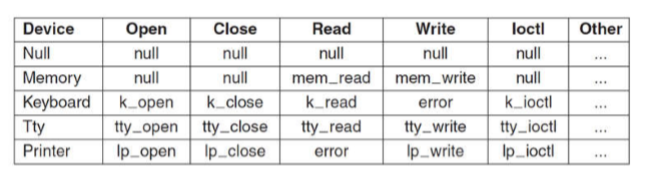
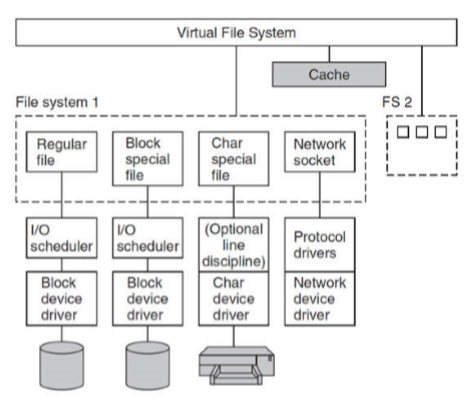
#### La struttura del kernel

Il kernel si trova direttamente in hardware e permette di abilitare le interazione coi dispositivi I/O e controllare l’accesso alla CPU, esso contiene i gestori degli interrupt, il principale metodo per interagire coi primi. Quando si verifica un interrupt, si interrompe il processo in esecuzione, si salva il suo stato e si abilita il driver appropriato alla periferica, questo dispatch è presente anche quando il kernel completa certe operazioni. Vi sono tre componenti principali indipendenti tra loro: la componente I/O, ovvero tutte le parti responsabili dell’interazione coi dispositivi e l’esecuzione delle relative operazioni, esse sono integrate nel VFS. Al livello più basso, tutte le operazioni I/O passano per i driver mentre sopra di essi vi è il codice del kernel (diverso per ogni dispositivo). Il software di rete è modulare e supporta diversi protocolli. Il livello più alto si occupa dell’instradamento, ovvero si preoccupa di mandare i pacchetti a destinazione, ritrasmettendoli in caso di perdite. Linux possiede tutte le funzionalità di un router nel kernel, anche se le prestazioni sono inferiori. Vi sono poi i driver dei dispositivi a blocchi, sopra di essi vi è lo scheduler dell’I/O che si occupa di ordinare e trasmettere le richieste ai dispositivi stessi. Nel livello più alto vi sono i file system, essi coesistono attraverso un’astrazione che nasconde le differenze delle architetture. Vi sono poi altre due componenti, essi si preoccupano di gestire memoria e processi mantenendo anche le mappature della memoria virtuale e una cache di pagine, oltre alla lettura in memoria su richiesta La più grande responsabilità della gestione dei processi è la loro creazione e distruzione, in più sceglie quale processo/thread avviare attraverso lo scheduler. I file system, per ottimizzare i tempi, copiano i file nella cache delle pagine quando vengono richiesti, altri possono essere persino creati dinamicamente. In più, quando bisogna liberare delle pagine, il sistema della memoria virtuale utilizza delle partizioni per effettuare dei backup di parti della memoria principale. Linux supporta il caricamento dinamico dei moduli, essi possono essere per aggiungere/sostituire i driver dei dispositivi di default. Infine vi è l’interfaccia in cui avvengono le system call, esse passeranno il controllo a uno dei componente descritti precedentemente. 

#### Processo d’avvio

All’avvio del computer, il BIOS esegue il cosiddetto POST (Power On Self Test) in cui rileva la presenza di dispositivi e ne fa l’inizializzazione, in seguito MBR (Master Boot Record), il primo settore del disco, esso avvierà un programma chiamato boot, il quale libererà spazio copiando se stesso in un indirizzo fisso di memoria alta. Liberato lo spazio, il boot carica il codice del bootsector della partizione segnata come attiva, esso leggerà la radice del file system per caricare il bootloader. Una volta eseguito, quest’ultimo caricherà la sua configurazione, nel caso ve ne siano differenti all’interno della macchina, sarà presentata una lista di configurazioni all'utente e verrà caricato quello scelto. Infine il boot loader carica il kernel il quale verrà avviato. Da qui in poi, inizia l’auto configurazione del kernel, inizialmente esaminerà quali dispositivi sono realmente connessi all macchina, aggiungendoli a una tabella apposita nel caso lo siano e caricando i relativi driver. Una volta configurato il sistema, si passa alla creazione del cosiddetto processo 0, esso programmerà il clock, monterà il file system e alla fine si sdoppierà in due ulteriori processi: init e il page daemon. Se quest’ultimo andrà a gestire la paginazione, Init controlla i suoi flag e avvierà uno o più terminali in base al numero di utenti della macchina. Per ciascun terminale, verrà eseguito Getty, un programma che digita “login:” e attende che l’utente inserisca un nome. Getty termina eseguendo login il quale chiederà una password all’utente, se essa corrisponde a quella memorizzata, login si sostituisce alla shell, altrimenti ne verrà richiesta un’altra.

#### I/O di Linux

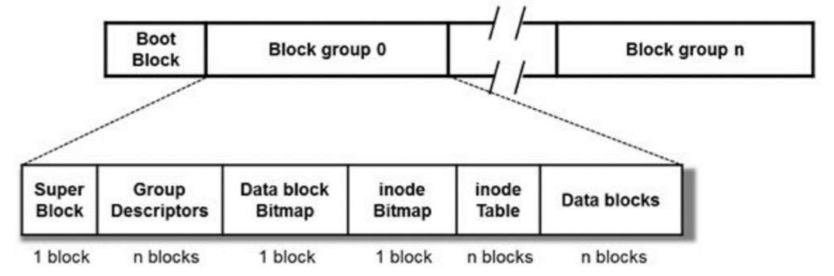
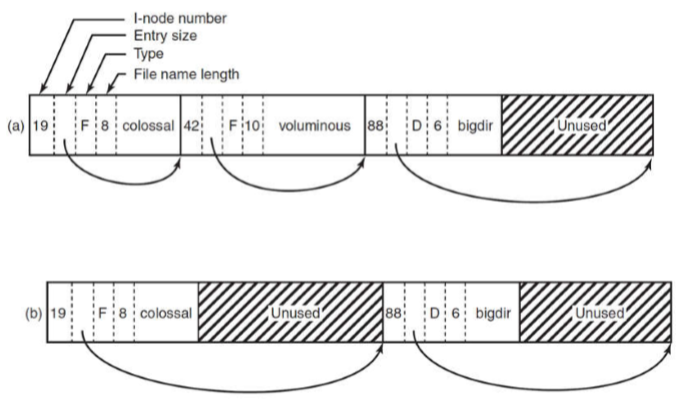
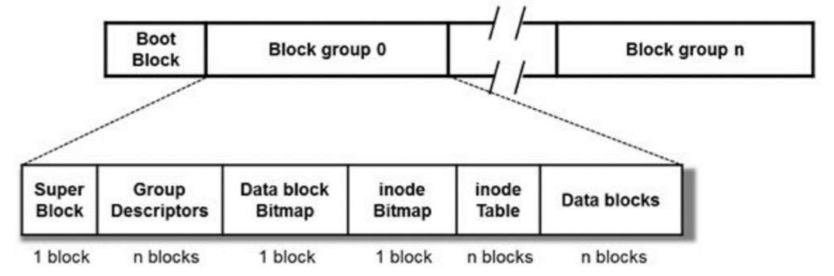
Linux permette l’accesso ai dispositivi I/O attraverso dei file speciali associati a essi con cui i programmi possono interagire. Questi file possono essere a blocchi o a caratteri, i primi possono indirizzare ogni blocco individualmente (infatti sono di solito utilizzati per i dischi) mentre i secondi sono utilizzati nelle periferiche che inviano/ricevono tramite stream e l’accesso si può fare solo sequenzialmente, in più hanno bisogno di ulteriori controlli non necessari nei primi. L’implementazione dell’I/O è formata da un insieme di driver, ognuno associato a un dispositivo, la loro funzione è quella di isolare l’hardware dal resto del sistema utilizzando apposite interfacce. Quando si accede a un file speciale, il file system determinerà il dispositivo associato, essi verranno anche utilizzati come indice nella tabella hash corrispondente al tipo(blocchi o caratteri). La tabella hash contiene puntatori a funzioni relative al dispositivo stesso, quando si effettua un'operazione su un file speciale, il sistema indicizza la tabella per selezionare la struttura appropriata e dopo chiamerà la funzione richiesta. Ogni operazione su file contiene un puntatore a una funzione contenuta nel driver corrispondente. Il driver è diviso in due parti, la parte superiore si interfaccia con Linux e viene eseguita in modo utente mentre quella inferiore viene eseguita in modo kernel e si interfaccia con la periferica. Il sistema di I/O è diviso in due parti: La gestione dei file speciali a blocchi e a caratteri. L’obiettivo dei primi è minimizzare il numero di trasferimenti da effettuare, per farlo si utilizza una cache posta tra i driver e il file system. Le versioni di Linux più recenti hanno una cache unificata a quella delle pagine in cui un livello di blocchi generico effettua le dovute traduzioni e permette le operazioni su di essi. Ogni volta che viene richiesto un blocco, si controlla se è presente in cache e si prende da lì in tal caso, altrimenti la cache stessa accede al disco, lo copia e lo invia all’utente, Per evitare permanenze troppo lunghe dei blocchi nella cache, tutti quelli modificati vengono scritti su disco ogni 30 secondi. Per ridurre al minimo la latenza della testina, vi è uno scheduler dell’I/O il cui obiettivo è quello di raggruppare/riordinare le richieste di operazioni nei dispositivi a blocchi, esso si basa sull’algoritmo dell’ascensore, esso ordina le richiesta in una lista doppiamente linkata e tutte quelle nuove vengono aggiunte in modo ordinato. Per evitare la starvation delle richieste, si aggiungono due ulteriori liste che gestiscono le operazioni di lettura/scrittura in base alle scadenze. Se una richiesta sta per scadere, esse viene gestita prima di tutte le altre, indipendentemente dall’ordine. Oltre ai file normali, esistono anche i cosiddetti raw, ovvero i file a blocchi di basso livello, essi permettono ai programmi di accedere al disco utilizzando numeri di blocco assoluti, senza considerare il file system, sono solitamente usati nella paginazione e nella gestione del sistema. L’interazione coi dispositivi a caratteri è semplice, l’unica eccezione riguarda la discipline-line, essa può essere associata a un dispositivo terminale per fare da interprete coi dati scambiati con esso. Se un processo vuole interagire con ciascun carattere, la discipline-line verrà impostata a raw, evitando così i controlli. 

L’interazione coi dispositivi di rete è differente dato che sono asincroni e quindi meno adatti all’interfaccia dei normali dispositivi a caratteri. I driver di questi dispositivi producono pacchetti che verranno in seguito instradati e passati all’applicazione, i processi interagiscono con essi attraverso i socket, vi sono anche i cosiddetti raw socket i quali permettono di accedere direttamente ai dispositivi di rete, tuttavia possono essere solamente creati dai superuser.

#### File system di Linux

Il primo file system utilizzato da Linux fu MINIX 1, esso però limitava nomi e dimensioni dei file rispettivamente a 14 caratteri e 64 MB dato che non era fatto per un utilizzo al di fuori di quello accademico. Un primo miglioramento vi fu col file system EXT, esso comprendeva nomi fino a 255 caratteri e una dimensione massima di 2 GB però era più lento del suo predecessore. La versione successiva, il file system EXT2, oltre a ingrandire la dimensione massima dei file, migliorò anche le prestazioni. Linux supporta dozzine di file system utilizzando il VFS. Un file di Linux è una sequenza di byte contenenti informazioni arbitrarie e i relativi nomi possono essere al massimo di 255 caratteri e sono limitati alla codifica ASCII (A eccezione di NULL). Molti programmi si aspettano che il formato dei nomi dei file sia il nome più l’estensione separati da un punto, il sistema operativo però non impone queste convenzioni. File e Directory possono essere specificati in due modi: il primo è il percorso assoluto, ovvero il percorso che bisogna fare a partire dalla radice fino ad arrivare al file/directory interessata, il secondo è il percorso relativo, il quale è analogo al primo ma esso parte dalla directory di lavoro, entrambi sono univoci nel file system a meno che non vi siano dei link.

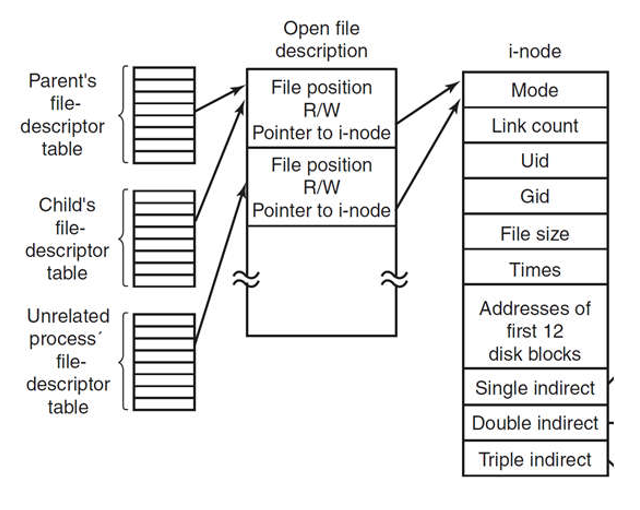
#### File system EXT2

Il file system EXT2 è un file system, ancora oggi in uso, che corregge gli errori dovuti alle prestazioni relativi alla versione precedente. Il primo blocco di EXT2 è il superblock, esso contiene informazione relative al layout del file system come numero di i-node e blocchi e l’inizio della free list, i successivi sono i group descriptor i quali contengono informazioni riguardanti la bitmap e il numero di blocchi liberi, i-node e delle directory presenti nel gruppo. I-node e blocchi liberi vengono tracciati attraverso due bitmap, ognuna lunga un blocco. Dopo le due bitmap vi è la parte dedicata alle i-node, ognuna è lunga 128 byte e descrive uno e un solo file, essi contengono informazioni di gestione e altre che permettono di individuare i blocchi. Infine vi è la parte dedicata ai blocchi di dati. Gli i-node corrispondenti a cartelle sono dispersi tra i gruppi di blocchi del disco. EXT” tenta di posizionare i file ordinari nel gruppo di blocchi corrispondente a quello della directory genitore, sempre che ci sia spazio sufficiente. Le bitmap vengono utilizzate per velocizzare le allocazioni di nuovi dati nel file system. Quando vengono allocati i blocchi di un nuovo file, EXT2 esegue una preallocazione di otto blocchi aggiuntivi per ridurre al minimo la frammentazione dovuta alle scritture successive. Le directory, come i file possono avere nomi di 255 caratteri, ognuna contiene delle voci, ognuna segue la successiva, che permettono di riferire file e ulteriori cartelle. ogni voce è formata da quattro campi di lunghezza fissa e uno variabile, i primi quattro sono rispettivamente il numero della i-node, la grandezza della voce in byte (necessario per trovare quella successiva), il tipo e infine la lunghezza del nome. Il campo variabile è il nome del file, estendibile fino a un massimo di 32 bit. Questa struttura è adatta per le directory piccoli, se fosse grande, il sistema ci metterebbe molto tempo a trovare la voce interessata dato che è lineare, quindi vengono utilizzati i B+ alberi per questo scopo. Il sistema utilizza anche una cache per ridurre i tempi di accesso alle directory, ogni directory entry viene inserita all'interno di essa per ciascuna componente del percorso e si cerca l’elemento successivo fino a quando non si raggiunge l’i-node del file. Quando è richiesto un file, si va a controllare se è presente in cache, se è così, viene preso da lì e viene messo nella tabella delle i-node, una struttura gestita dal kernel. Dato che la dimensione dei gruppi è omogenea, per trovare una i-node dato l’indirizzo, si procede nel seguente modo:

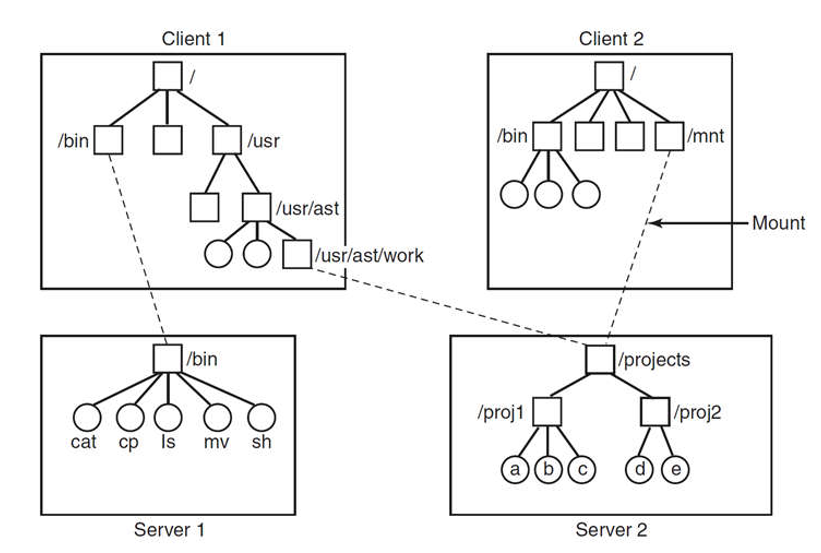
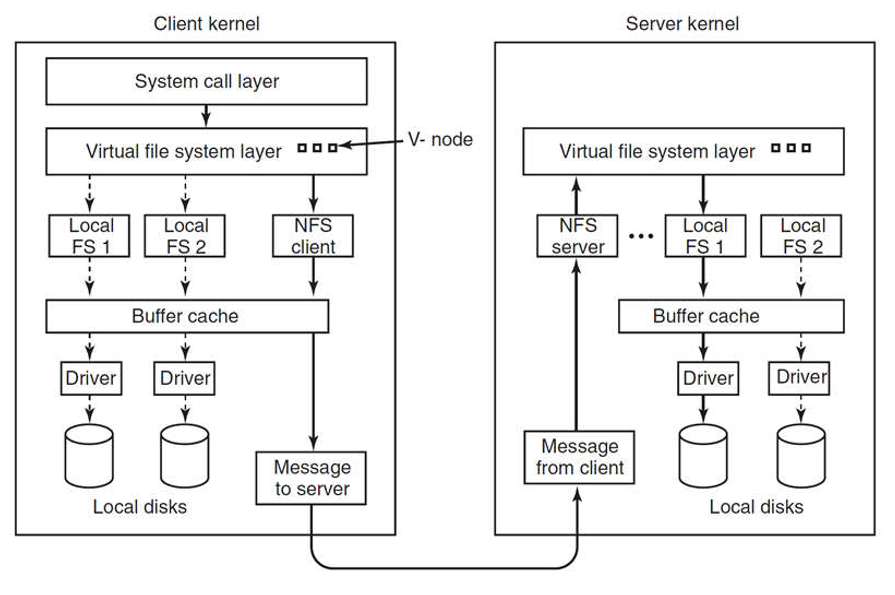
* Trovo a quale gruppo appartiene: group=(indirizzo-1)/#i-node;
* Trovo l’indice della i-node richiesta: index=(indirizzo-1)%#i-node;
* Vado a prendere l’i-node corrispondente all’indice index\*dimTable nella tabella delle i-node.

L’i-node di un file Linux è formato dai seguenti campi:

* Mode: indica tipo e protezione dei file;
* NLinks: indica il numero di directory entry che lo puntano;
* Uid: indica l’id dell’utente proprietario;
* Gid: indica l’id del gruppo proprietario;
* Size: indica la dimensione del file in byte;
* Address: indica gli indirizzi dei primi 12 blocchi diretti e i 3 indiretti;
* Gen: indica il numero di utilizzi della i-node;
* Atime; indica l’ultimo accesso;
* Mtime: indica l’ultima modifica;
* Ctime: indica l’ultima volta che l’i-node è stata cambiata.

Quando si effettua una system call, come una lettura, il kernel prende il controllo e parte dai parametri dati e dalle informazioni delle sue tabelle interne riferite all’utente, una delle voci di queste tabelle è l’array di file descriptor in cui vengono riferiti tutti i file aperti. Associato a ogni file descriptor vi è una posizione, essa indica a quale byte inizia l’operazione successiva, essa può essere inserita direttamente nella i-node ma fallirebbe perché può capitare che due o più processi accedano allo stesso file nello stesso momento.un’altra soluzione è mettere la posizione nella tabella dei file descriptor in modo che ogni processo che accede al file abbia ne abbia un’istanza privata, questa soluzione però non funziona perchè, se vi sono due comandi da eseguire in ordine, al termine del primo non deve esserci 0 nella posizione iniziale ma il valore con cui è terminato. Per risolvere questo problema è stata introdotta la tabella globale dei file descriptor aperti in cui inserire la posizione, in questo modo i processi figli possono ereditare la posizione del file dal genitore senza conoscerla direttamente, quelli non correlati che aprono lo stesso file ne avranno una propria che non andrà a disturbare le altre.

#### Network File System (NFS)

Il network file system è un insieme di meccanismi che permette di collegare più file system di più macchine in un unico grande file system logico. Esso permette ai calcolatori stessi di essere sia client che server, ogni server infatti esporta una o più cartelle (e di conseguenza anche tutte le sottocartelle) con cui i client faranno accesso da remoto quando effettueranno un mount, esse diventeranno parte della struttura gerarchica. I punti di mount sono solamente locali al client, il server non sa nulla su dove viene eseguito. Per una corretta gestione, NFS specifica due protocolli, il primo si occupa delle operazioni di mount, i client possono inviare pathname a un server e richiedere il permesso di mount sulla directory richiesta. Se il pathname è legale e la directory è esportata, il server restituisce un handle al client, esso contiene campi che identificano in modo univoco il file system del server. La maggior parte delle versioni di Linux supporta l’automounting, esso consente a un insieme di directory remote di essere associati a una reale, in questo caso il client non esegue alcun mount al suo avvio, tuttavia alla richiesta di un file remoto il sistema operativo invia un messaggio a tutti i server ed esegue il mount col primo che risponde. I vantaggi dell’automounting sono che, se il server non è disponibile, non vi è possibilità di avviare il client, in più se un client non ha la necessità di connettersi a un server in quel momento, non spreca risorse per farlo, questo consente ai client di cercare un insieme di server in parallelo, raggiungendo così un certo grado di tolleranza ai malfunzionamenti e migliorando le prestazioni. Il secondo protocollo NFS gestisce l’accesso a file e directory, i client infatti inviano massaggi ai server per leggere/scrivere file o manipolare le cartelle. Non è richiesta l’apertura di un file prima di un'eventuale lettura/scrittura, infatti il client invia al server un lookup contenente il nome del file, il server lo cercherà e restituirà l’handle. Il vantaggio di questo metodo è che il server non deve ricordare nulla sulla connessioni aperte coi client, questo approccio è detto stateless. NFS, come tutti i file system di Linux, si appoggia a VFS, esso mantiene una tabelle contenente voci che corrispondono a file aperti, i v-node. Una v-node può indicare se un file è remoto o meno, nel caso lo sia, vengono fornite informazioni per l’accesso, altrimenti viene registrata l’i-node insieme al rispettivo file system. Il mount di un file system remoto viene eseguito specificando la directory remota, esso analizza il nome e scopre dov’è posizionata nel server e infine lo contatta per richiederne l’handle. Il kernel costruisce una v-node e richiede del codice del client remoto per creare una r-node (una i-node remota) in cui conservare l’handle ricevuto. Quando un file remoto è aperto, il kernel vede che la directory è remota, quindi chiede al client NFS di aprire il file e cercare la parte restante del percorso sul server, ottenendo un handle. Verrà poi creato un r-node per il file remoto e lo si comunica al VFS, il quale inserirà nelle sue tabelle un v-node che punta all’i-node remota. Al chiamante verrà poi dato un file descriptor per il file remoto. I trasferimenti tra client e server vengono effettuati in blocchi di solito grandi 8KB per motivi di efficienza. Quando vengono utilizzate system call su un file descriptor, il VFS determina se il file è remoto o meno e, nel caso lo sia, invia un messaggio al server contenente l’handle, l'offset e il numero di byte del file. Il server, quando ha ricevuto, lo dà al suo VFS il quale lo cerca e lo restituisce al client.Dopo che il client ha ricevuto i dati, riceverà il pezzo successivo in modo da essere disponibile a breve, questo approccio è detto read ahead e migliora le prestazioni in modo considerevole. I trasferimenti tra processi sono analoghi a quelli client- server con la differenza che, se un blocco è minore di 8 KB, i dati vengono accumulati e trasferiti quando il blocco è pieno, in più quando un file viene chiuso, tutti i dati vengono immediatamente inviati al server. Per evitare troppi accessi al disco, i server dispongono di una cache invisibile ai client, in essa vengono messi i dati. I client invece gestiscono due cache, una per gli attributi e l’altra per i dati dei file, esse migliorano le prestazioni ma fanno sorgere ulteriori problemi riguardanti la memorizzazione dei blocchi. Se due client memorizzano in cache lo stesso blocco e uno di essi lo modifica, l’altro conterrà una copia scaduta del blocco stesso, per evitare ciò si utilizza un timer associato a ogni blocco, quando esso scade, viene eliminata la relativa voce. Ogni volta che viene aperto un file meorizzato in cache, si invia un messaggio al server per scoprire quando è stata fatta l’ultima modifica, se è essa non corrisponde a quella in cache, viene presa la copia nel server. Per evitare ulteriori problemi, ogni 30 secondi tutti i blocchi modificati nella cache vengono inviati al server.

### File System NT

NTFS (New Technology File System) è il file system di default di Windows dalla versioni NT a oggi e permette di migliorare la sicurezza e le funzionalità del suddetto sistema operativo, esso utilizza indirizzi da 64 bit, permettendo il supporto a partizioni grandi 2^64 byte al massimo, anche se si limitano a dimensioni più piccole per altre considerazioni. I nomi dei file sono case-sensitive e limitati a 255 caratteri mentre i percorsi assoluti nel possono contenere 32767, essi sono in Unicode, permettendo così di scrivere nella propria lingua nel caso non venga utilizzato l’alfabeto latino. I file non sono solo sequenze lineari di byte, essi infatti consistono di molteplici attributi, ognuno rappresentato da un flusso. Ogni flusso ha un nome nel seguente formato: NomeFile:NomeFlusso, ognuno ha una sua dimensione e può essere chiuso indipendentemente da tutti gli altri, in più sono anche utilizzati per contenere i metadati di un file, tuttavia sono delicati e frequentemente vengono cancellati quando si trasportano i file in altri file system, sulla rete o quando si ripristina un backup. NTFS è un file system gerarchico, il separatore è ”\” e i concetti di directory attuale (.) e directory genitore (..) sono gli stessi dei file system di UNIX, anche se implementati come convenzioni. Gli hard link e il controllo di attraversamento sono implementati ma solo per il sottosistema POSIX. NTFS supporta anche i link simbolici, la loro creazione però è riservata agli amministratori per evitare problemi di sicurezza, sono implementati utilizzando i cosiddetti punti di reparse.

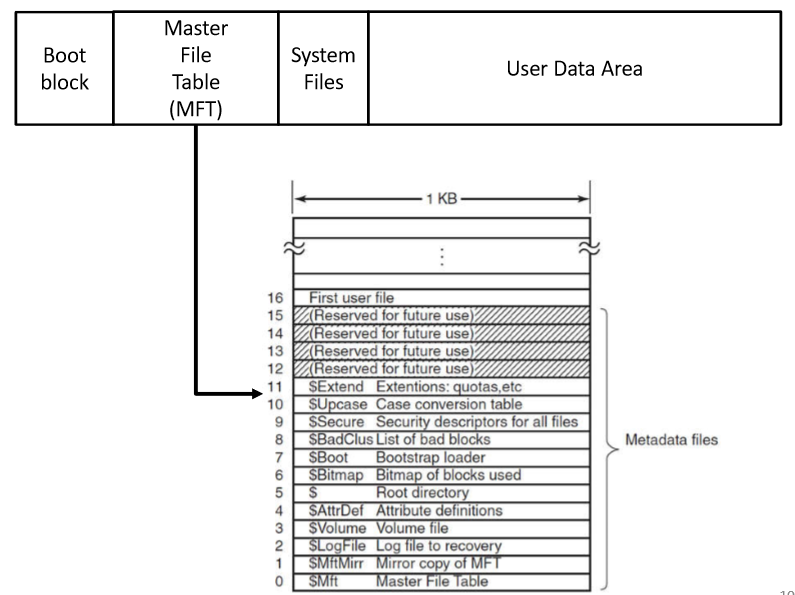
#### Journaling

NTFS supporta due meccanismi che permettono di rilevare modifiche a file/directory: il primo è un’operazione I/O chiamata NtNotifyChangeDirectoryFIle, il quale passa al sistema un buffer che verrà restituita quando viene rilevata una modifica a una directory o a un suo sottoalbero, esso conterrà una lista di record delle modifiche. Il secondo meccanismo invece mantiene un elenco di record di modifiche in un file detto change journal, di solito è molto grande è vi è poca probabilità che le voci vengano riutilizzate prima che possano essere esaminate.

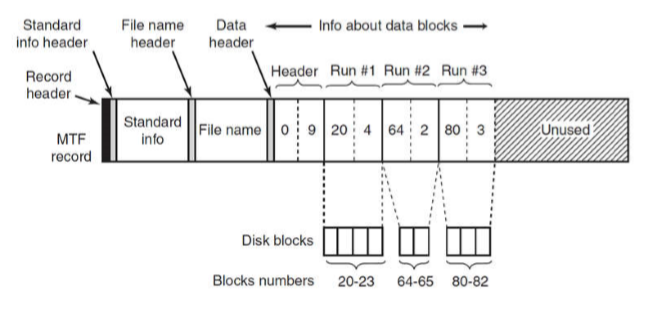
#### Crittografia

Windows fornisce due opzione per permettere la crittografia di file e cartelle, il primo è contrassegnare le cartelle come crittografati in modo da criptare i file all’interno e quelli che verranno inseriti in futuro, queste operazione vengono gestite dal un driver detto EFS (Encryption File System). Il secondo metodo è l’utilizzo del BitLocker, il quale codifica quasi tutti i dati su un volume.

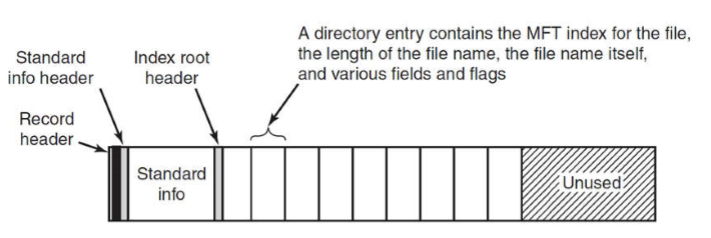
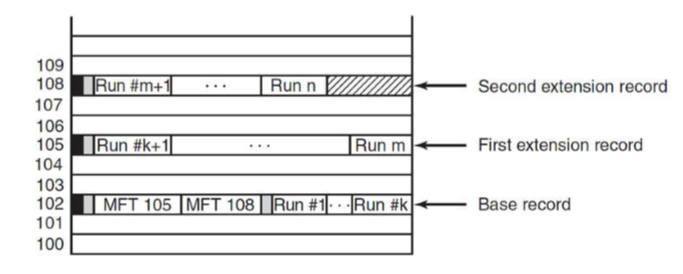
#### Struttura

Ogni volume NTFS è una sequenza lineare di blocchi di dimensione fissa e vengono riferiti attraverso un offset, la struttura dati principale è la MFT (Master File Table), una sequenza lineare di record lunghi 1KB, ogni record descrive un solo file e una sola directory e contiene gli attributi e la lista degli indirizzi dei blocchi. Nel caso un file sia troppo grande per un solo record, ne vengono utilizzati molteplici. La MFT è un file che può essere posizionato in qualsiasi parte del volume, eliminando così il problema dei settori difettosi nella prima traccia, essa può crescere fino a una dimensione massima di 2^48 blocchi. Ogni record consiste in una sequenza di coppie (header attributo, valore), ciascun attributi inizia con un header che indica il tipi di attributo e la lunghezza del valore (alcuni di essi possono avere lunghezza variabile, se è troppo lungo, lo si posiziona da qualche parte nel disco). I primi 16 record della MFT sono riservati ai metadati di NTFS (infatti il loro nome inizia per $):

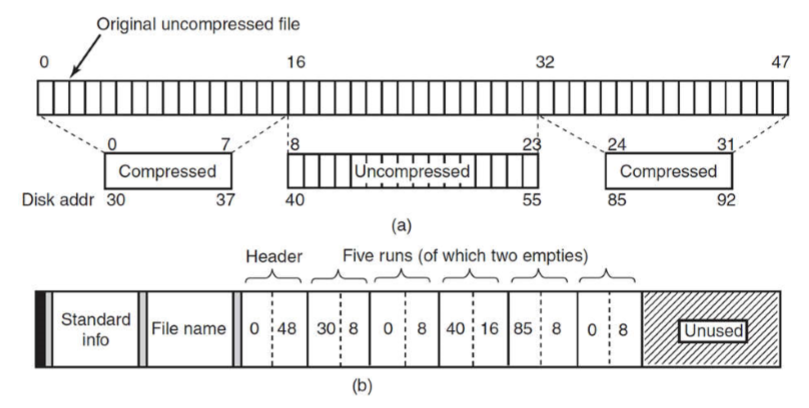
* il primo descrive la MFT stessa, indicando dove sono posizionati i relativi blocchi e viene anche riferito nel blocco di avvio.
* Il record 1 è un duplicato della prima parte di MFT;
* il record 2 è un file log, in esso vengono registrate le azioni prima della loro esecuzione in modo da poter ripristinare il file in caso di malfunzionamenti.
* il record 3 contiene informazioni riguardanti il volume;
* il record 4 definisce gli attributi dei record;
* il record 5 contiene un riferimento alla radice dell’albero del file system;
* il record 6 contiene la bitmap, la quale gestisce lo spazio libero;
* il record 7 punta a un file che carica la sequenza di istruzioni iniziali;
* il record 8 gestisce i settori danneggiati connettendoli tra loro, assicurandosi che non siano mai presenti nei file;
* il record 9 contiene informazioni riguardanti la sicurezza;
* il record 10 permette la mappatura di maiuscole e minuscole;
* il record 11 riferisce a una directory che contiene vari file utili al file system;
* i record dal 13 al 15 sono riservati per usi futuri.

NTFS definisce 13 attributi che possono comparire nei record MFT, essi seguono direttamente i loro header e, se superano 24 byte, vengono messi in un blocco a parte (attributo non residente). In più possono essere ripetuti ma tutti devono apparire nell’ordine prefissato da MFT. Le informazioni standard contengono vari dati, tra cui il proprietario e le informazioni di sicurezza, utili sia al file system che all’utente stesso. La lista degli attributi è richiesta nel caso in cui essi non riescono a stare nel record MFTe indica dove trovare gli extension record. Ogni voce della lista contiene un indice di 48 bit che indica dove si trova l'extension record cui ha associato un numero di 16 bit per consentirne la verifica. I file NTFS hanno un ID simile alla i-node del file system di UNIX, con esso è possibile aprire i file ma non sono sempre utili quando resi persistenti, dato che si basa su un record che può cambiare quando spostato. Il punto di reparse indica a un meccanismo in grado di montare esplicitamente file system e link simbolici, esso è formato da 5 attributi: i primi due identificano il volume mentre gli altri tre si occupano di implementare le directory, si usano le liste per quelle piccole e i B+ alberi per quelle grandi. Il più importante di tutti gli attributi è il flusso, esso può essere presente più volte in un file NTFS. Di default, il flusso non ha un nome ma, nel caso ci fosse, è memorizzato nell’header dell’attributo. Dopo l’header vi è un elenco di indirizzi che indicato tutti i blocchi associati al flusso, al posto di esso potrebbe anche esserci il flusso stesso.

#### Allocazione della memoria

Per questioni di efficienza, i blocchi vengono tracciati e assegnati in serie consecutive, questo metodo è raggiungibile allocandone diversi alla volta, quando possibile. I blocchi in un flusso sono descritti da una sequenza di record, ognuna descrive logicamente una sequenza, il numeri di record è direttamente proporzionale al numero di buchi tra le sequenze. Questo tipi di file è detto file sparso, ogni record inizia con un header che contiene l’offset del primo nel flusso e quello del primo non coperto dal record. In seguito vi sono delle coppie di numeri, ognuna rappresentante una sequenza, il primo numero è l’indirizzo del primo blocco mentre il secondo è il numero di blocchi della sequenza. Questo modello non ha un limite massimo per quanti riguarda le dimensioni, se non quelle fisiche del disco stesso, è comunque disponibile un metodo che permette di ridurre la dimensione delle coppie sotto i 16 byte omettendo i byte più significativi, dato che sono a 0. Per rendere la ricerca dei blocchi più efficiente, vengono utilizzate le ridondanze, basta solamente esaminare gli header dei record e non le coppie. Quando un record MFT finisce lo spazio che ha a disposizione, ne vengono utilizzati altri, in questo modo è possibile memorizzare file grandi e frammentati. I record possono anche contenere un certo numero di voci di directory, ognuna è di lunghezza fissa la quale indice l’indice di una voce MFT di un file, la lunghezza del suo nome e altri campi e flag (Questo vale per le directory piccole, quelle grandi sono implementate come B+ alberi). Il parsing di un percorso in NTFS inizia dalla directory principale, ogni nome viene cercato singolarmente per trovare il relativo indice MFT, quando viene trovato il file richiesto, NTFS imposta dei puntatori ai suoi metadati e viene passato al gestore dell’I/O, un volta aperto, viene chiamata una IoCompleteRequest che passa l’IRP ai gestori dell’I/O e,alla fine, viene messo un handler per il file nella tabella degli handle del processo attuale. NTFS supporta anche gli hard link e i link simbolici, questi ultimi utilizzano i punti di reparse, essi vengono interpretati dal gestore di oggetti come un nome di percorso con cui fare il parsing e le operazioni I/O. 

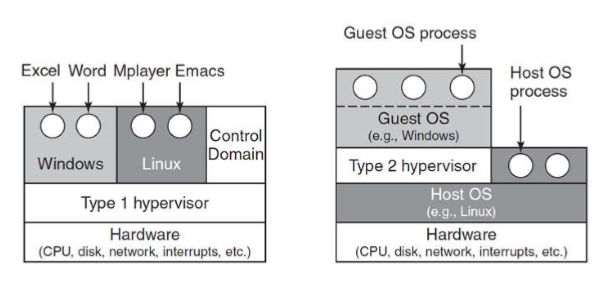
#### Compressione dei file

NTFS supporta la compressione trasparente di file e cartelle, essa funziona nel seguente modo: vengono esaminati i primi 16 blocchi logici del file (senza considerare quante serie occupano) e viene avviato l’algoritmo di compressione, se il risultato è minore o uguale a 15 blocchi, i dati vengono scritti su disco, altrimenti vengono lasciati così come sono. In seguito vengono presi i successivi 16 blocchi e così via. Alla rilettura del file bisogna identificare quali parti sono compresse e quali, NTFS li distingue nel seguente modo:le coppie successive all’header hanno 0 come primo numero se sono compressi, altrimenti hanno il primo blocco della serie. L’accesso casuale a file compressi è possibile grazie a un complesso processo, bisogna infatti legge e decomprimere l’intera serie per poi passarlo al processo che lo deve leggere. La scelta di utilizzare 16 blocchi è infatti un compromesso, dato che numeri più piccoli l’avrebbero resa meno efficaci mentre quelli più grandi avrebbero causato un accesso casuale più costoso. 

### Virtualizzazione

La virtualizzazione è un metodo che permette a una macchina di ospitare più sistemi operativi emulando l’hardware. Queste emulazioni vengono dette macchine virtuali e devono funzionare esattamente come una macchina reale, il compito di garantire questo è affidato all’hypervisor e deve garantire:

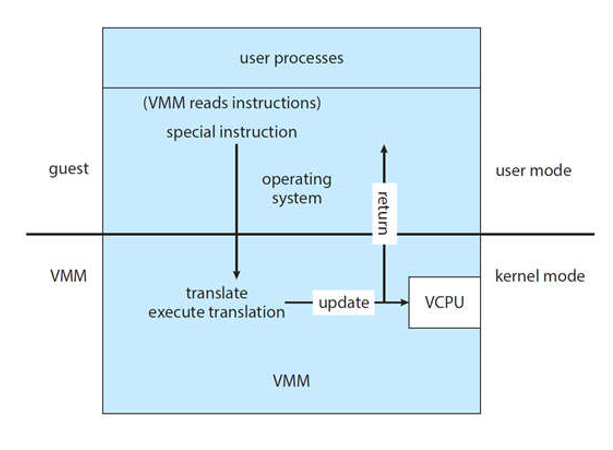
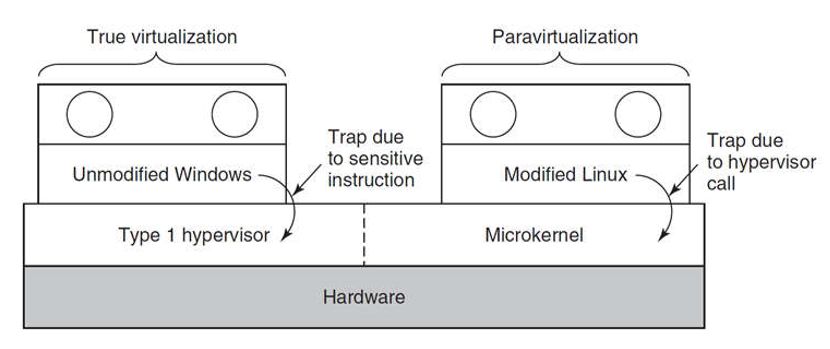
* Sicurezza: il controllo completo delle risorse virtualizzate;
* Equivalenza: il comportamento di un programma in una macchina virtuale deve essere identico a quello in una macchina reale;
* Efficienza: il codice dovrebbe funzionare senza eventuali interventi dell’hypervisor.

Un modo sicuro per eseguire le istruzioni è elaborarle tutte utilizzando un interprete, alcune possono essere eseguite direttamente, altre invece bisogna “ingannarle” con esso. Tra queste istruzioni vi sono le sensitive, ovvero quelle che si comportano in modi differenti a seconda della modalità (utente o kernel), e quelle privilegiate, quelle che causano trap se chiamate in modo utente. Un sistema per essere virtualizzabile deve avere l’insieme delle istruzioni sensitive come sottoinsieme di quelle privilegiate, in questo modo l’hardware solleverebbero trap, purtroppo questa qualità non è presente nei processori x86 dato che vi sono istruzioni sensitive che possono leggere stati sensibili senza causare trap. Questo problema fu risolto nel 2005 col supporto hardware per la virtualizzazione da parte dei produttori, nello specifico VT (Virtualization Technology) per Intel e SVM (Secure Virtual Machine) per AMD. All’avvio di sistema operativo, esso continua a funzionare fino a quando non viene sollevata una trap, la quale sarà gestita dall’hypervisor. Tutte le operazioni che causano trap sono controllate dall’hypervisor attraverso una bitmap, queste estensioni rendono possibile l’approccio dettato in precedenza, il trap-and-emulate. Prima del supporto hardware, gli hypervisor riscrivevano parti di codice del sistema operativo che potevano causare problemi, sostituendole con parti sicure che emulavano l’istruzione originaria, questo approccio è detto traduzione binaria e permette di sostituire le istruzioni sensitive ma non quelle privilegiate. Non è comunque necessario riscrivere tutte le istruzioni sensitive dato che, anche se hanno comportamenti differenti in modo kernel, non costituisce un problema in modo utente, per quelle che sono anche privilegiate si utilizza il trap-and-emulate. Esiste un’altra tecnica, detta paravirtualizzazione, diversa da quella completa perchè non ha lo scopo di creare una macchina virtuale identica all’hardware sottostante, essa presenta un’interfaccia software contenente delle hypercall, delle funzioni che consentono al guest di inviare richieste all’hypervisor e vengono usate per svolgere operazioni privilegiate e sensibili. L’unico problema è che il guest deve conoscere le API della macchina virtuale, di conseguenza è necessario personalizzarlo per un determinato hypervisor. Un hypervisor può essenzialmente essere di due tipi:

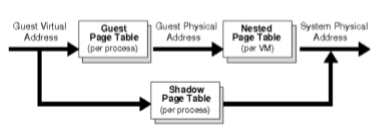
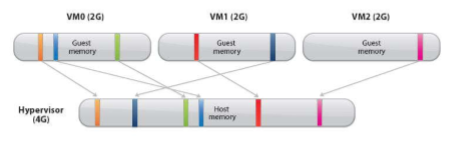
* Gli hypervisor di tipo 1 sono molto simili a un sistema operativo, dato che è l’unico programma in esecuzione nella modalità più privilegiata, il suo compito è supportare più macchine virtuali;
* Gli hypervisor di tipo 2 sono normali programmi che si basano sulle risorse del sistema operativo ospite.

Entrambi i tipi devono eseguire le istruzioni macchina in maniera sicura.

#### Virtualizzazione della CPU

Una macchina virtuale viene gestita come un processo in modo utente, di conseguenza non è consentita l’esecuzione di istruzioni sensitive e il suo sistema operativo crede di girare in modo kernel (anche se in realtà non lo è), questo approccio è detto modalità kernel virtuale. Cosa succede quando un sistema guest esegue un'istruzione consentita solo in modo kernel? Se la CPU è senza VT, essa fallisce e il sistema operativo si blocca, altrimenti viene generata un trap sull’hypervisor, esso andrà a controllare se la richiesta è partita dal sistema operativo o da un programma utente, nel caso fa in modo che venga eseguita mentre nel secondo ne emula il comportamento. I processori x86 hanno supportato quattro modalità di protezione (dette anche anelli), i programmi utente si trovano all’anello 3, quello coi privilegi più bassi, mentre all’anello 0 vi è il kernel, gli anelli 1 e 2 possono essere utilizzati dall’hypervisor. Quando viene utilizzata un’istruzione privilegiata, viene sollevata una trap sull’hypervisor il quale effettuerà controlli d’integrità prima di eseguirla, per quanto riguarda le istruzioni sensitive, lo stesso hypervisor riscrive il codice in un blocco di base e termina con un branch (non contiene ulteriori istruzioni che modificano il flusso). Appena prima di eseguire il blocco, l’hypervisor controlla se è presente un’istruzione sensitive e, se è così, la sostituisce con una chiamata di procedura dell’hypervisor, il branch viene anch'esso sostituito con una chiamata all’interno dell'hypervisor stesso in modo da ripetere l’operazione col blocco successivo, questo approccio è detto traduzione binaria. Per evitare di ripetere le traduzioni, i blocchi tradotti vengono inseriti in cache. Un’altra possibilità consiste nell’effettuare la traduzione binaria su tutto il codice del sistema gest sull’anello sostituendo tutte le istruzioni che causano trap in modo da garantire prestazioni migliori. Le trap sono infatti sono esose da gestire indipendentemente dal tipo di hypervisor utilizzato ed emulare le operazioni che li generano è anch’essa dispendiosa, una soluzione è la riconfigurazione del sistema in modo da rimuovere quello ospite e quindi concedere tutto lo spazio di indirizzi, tuttavia questo è impossibile nello spazio utente. Molti hypervisor di tipo 2 hanno un modulo kernel all’anello 0 in modo da manipolare l’hardware con le istruzioni privilegiate, in più permettono ai sistemi guesto di avere accesso all’intero spazio d’indirizzi, tuttavia a un certo punto devono poter ripulire il tutto e ripristinare lo stato precedente del processore. Per quanto riguarda la gestione degli interrupt, un hypervisor di tipo due deve riconfigurare completamente l’hardware per poter eseguire il codice del sistema host dato che si appoggia ai driver di quest’ultimo, quest’approccio è detto commutazione di mondo. Dato che l’approccio trap-and-emulate genera molti dispendiosi trap che riempiono la cache della CPU e tutte le strutture adiacenti a esso, per questo motivo alcuni hypervisor sostituiscono le istruzioni sensitive con delle hypercall al fine di migliorare le prestazioni. La gestione degli interrupt viene fatta attraverso una struttura dati in cui si tiene traccia di ogni sistema guest attraverso un interrupt flag, in questo modo se è richiesta la disattivazione degli interrupt da parte di un guest, l’hypervisor li disattiva solo su esso. Con la paravirtualizzazione, l’hypervisor definisce un’interfaccia di hypercall, essere verranno eseguite al posto delle normali istruzioni sensitive, comportandosi in questo modo come un vero e proprio microkernel. L’emulazione delle istruzioni da parte dell’hypervisor è data dal fatto che o il codice sorgente del sistema guest non era disponibile oppure esistono numerosi varianti. Nella virtualizzazione completa infatti, le istruzioni che causano trap vengono emulate dall’hypervisor mentre, nel caso della paravirtualizzazione, le trap vengono sollevate dalle hypercall. Come vengono eseguite eseguire le hypercall sull’hardware nativo e come si fa a modificare il kernel in modo da essere eseguibile con più hypervisor? Il kernel viene modificato in modo che chiami procedure speciali all’esecuzione di istruzioni sensitive, esse formano uno strato a basso livello, detto VMI (Virtual Machine Interface), tra l’hardware e l’hypervisor stesso, in più sono progettate in modo generico e non legate a una particolare piattaforma/hypervisor. Quando il VMI è in funzione direttamente sull’hardware, viene controllato con una libreria che fornisce le istruzioni effettive mentre, se è su un hypervisor, il sistema guest è collegato a diverse librerie le quali eseguiranno le hypercall adatte. 

#### Virtualizzazione della memoria

La mappatura della memoria virtuale è definita mediante tabelle di pagine multilivello che di solito vengono eseguite in modo che il sistema operativo imposti un registro di controllo nella CPU in modo che punti alla tabella superiore. Nel caso del macchine virtuali si utilizza un’istruzione sensitive la quale può generare una trap oppure la chiama di un hypercall. Come vengono allocate le pagine? Una prima soluzione è allocare le pagine fisiche per una macchina virtuale e impostare le tabelle in modo che mappino quelle virtuali, vi è però un problema: se una seconda macchina virtuale chiede la mappatura sulle stesse pagine fisiche, l’hypervisor non saprebbe più come andare avanti dato che sono già utilizzate, può prendere delle pagine libere ma dovrebbe prima creare una tabella che mappi le pagine virtuali su quelle nuove. Questo approccio è complesso e oneroso da gestire, quindi occorre utilizzare un altro metodo: per ogni macchina virtuale, l’hypervisor crea una shadow page table in cui vengono mappate le pagine virtuali su quelle effettive. Se una macchina virtuale modifica la sua page table, l’hypervisor dovrà cambiare la sua shadow page table e, in più, i sistemi guest non utilizzano operazioni sensibili per cambiarle, di conseguenza l’hypervisor non si accorge di nulla e non viene a conoscenza della modifica. Una possibile soluzione è far tenere traccia all’hypervisor della pagina della memoria virtuale contenente la tabella superiore, essa può essere ottenuta quando il sistema guest carica la tabella nel registro per la prima volta dato che l’istruzione è sensibile. A questo punto l’hypervisor può creare la shadow page table e mappare la tabella superiore e tutte quelle a cui punta. Un altra soluzione è fare il contrario, ovvero quando l’hypervisor permette di aggiungere qualsiasi mappatura alle sue tabelle, nella shadow page table non cambia niente e l’hypervisor stesso non se ne accorge. Quando però un sistema guest cerca di accedere a qualsiasi pagina nuova, si verifica un errore e il controllo passa all’hypervisor, il quale verifica il contenuto delle tabelle per vedere se deve aggiungere una mappatura, aggiungendola e rieseguendo l’istruzione in caso positivo. La gestione delle shadow page table ha portato i produttori ad aggiungere a livello hardware le nested page table, che tendono a eliminare overhead gestendo le page table a livello hardware. L’hardware esamina queste tabelle per trovare l’indirizzo fisico corrispondente a quello virtuale. Grazie alla virtualizzazione, è possibile allocare più macchine virtuali anche la RAM richiesta supera in capacità quella reale, essa infatti viene fornita dall’hypervisor quando necessario attraverso una tecnica detta ballooning. Il ballooning utilizza un driver speciale che comunica con l’hypervisor, il quale può allocare(deallocare pagine su richiesta di quest’ultimo. Con l’aumentare delle richieste,aumenta anche la scarsità di memoria del sistema guest, per evitare ciò il guest stesso fa lo swapping delle pagine meno importanti, quando il driver “si sgonfia”, esso può allocare più memoria perchè nel frattempo si è liberata. Per evitare il page out delle pagine del processo,si esegue il Pin DOwn, ovvero si marcano le pagine come non paginabili. L’hypervisor monitora sempre il page fault d’ogni processo e, nel caso ve ne siano troppi, fornisce pagine per evitare il thrashing.

#### Virtualizzazione dell’I/O

Cosa deve fare l’hypervisor quando i sistemi guest causano trap per rilevare le periferiche? Un primo approccio è informare le periferiche stesse per sapere chi è effettivamente connesso e, di conseguenza, i sistemi guest caricheranno i relativi driver in modo da poterli utilizzare. Le operazioni I/O sono sensibili e causano trap, di conseguenza l’hypervisor si preoccupa di leggere/scrivere sulle periferiche stesse. Vi è però un problema, i sistemi guest ritengono di avere il controllo su un’intera partizione del disco, per risolverlo l’hypervisor crea file/regione del disco e li assegna a ogni macchina virtuale. Ogni sistema gues può convertire il numero di blocchi a cui ha accesso in un offset nel file/regione del disco utilizzati per la memorizzazione e per l’esecuzione dell’I/O. Un altro problema dell’I/O è l’uso della DMA, essa infatti utilizza indirizzi di memoria assoluti, di conseguenza l’hypervisor deve intervenire rimappando gli indirizzi in memoria attraverso delle trap. Per evitare l’inefficienza delle trap, è possibile utilizzare l’I/O MMU, un componente che utilizza page table per mappare sugli indirizzi fisici un indirizzo di memoria che un dispositivo vuole utilizzare, l’hypervisor imposterà le page table in modo che una periferica non utilizzi memoria che non gli appartiene. Nelle macchine virtuali, le I/O MMU hanno numerosi vantaggi:

* il passthrough di dispositivi consente l’assegnazione di un indirizzo fisico a una macchina virtuale per mantenere lo stesso spazio di indirizzi. In più, l’I/O MMU consente la rimappatura degli indirizzi in maniera trasparente;
* L’isolamento dei dispositivi assicurano che essi accedano alla loro macchina virtuale senza mettere a rischio l'integrità delle altre;

La DMA e gli indirizzi non rappresentano l’intera questione degli I/O, infatti occorre virtualizzare anche gli interrupt in modo da farli arrivare alla macchina giusta quando vengono generati, l’I/O MMU per risolvere questo problema, supporta la rimappatura degli interrupt.