1) ENTRADA E SAÍDA

A ideia é mandar dados e o dispositivo fazer alguma coisa com esses dados, por exemplo, o disco grava, a impressora imprime. Apenas o SO que roda em modo núcleo pode mandar o dispositivo parar de funcionar ou não. Por segurança, só o SO consegue executar instruções de IN e OUT para o dispositivo, ou seja, apenas ele se comunica diretamente com o dispositivo.

A comunicação entre CPU e dispositivo é feita através de instruções especificas de E/S. Ex.: IN e OUT. Os operadores significam: IN → Identificador do dispositivo onde será colocado o dado
OUT → identificador do dispositivo do dado que será enviado

Na Intel a identificação do dispositivo é feita pelo Nº de porta e o Registrador:

IN AH, 3F8H
OUT 3F8H, BH

Onde AH e BH são registradores, 3F8 o Nº da porta e o H (de 3F8H) indica que o nº está em hexadecimal.

Intel: Interrupções de E/S – IN/OUT.
Portas de E/S: Porta Serial - 2F8H – 2FFH e Porta Paralela - 378H – 37FH.

Exemplo de entrada: receber dados da rede via placa de rede
Exemplo de saída: exibição de informação na tela

Na INTEL quando está enviando é OUT, quando está recebendo é IN cada instrução deve-se identificar o dispositivo que está realizando a comunicação e o dado, no caso de OUT. E onde o dado vai ser colocado no caso de IN.

No IN tem que ter alguma coisa que identifica o dispositivo aonde vai ser alocado o dado e no caso de OUT vai ter o identificado do dispositivo e qual o dado que vai ser enviado. Por exemplo, no caso da INTEL, uma instrução de IN, no primeiro operando vai ter a porta, do registrador que vai receber o dado, depois o numero de porta (em assembly).

Alguns dispositivos no caso dos computadores tem um numero de portas para dados, por exemplo, teclado tem um numero de portas para comunicar com eles. Cada dispositivo padrão tem a sua porta com registrador de dados e através dessa porta que o SO se comunica com esses dispositivos.

Existem dispositivos que não são padronizados, pois surgiram depois do dispositivo padrão e terão que utilizar dispositivos não padronizados. Nesse caso o SO não vai estar pronto para se comunicar de primeira com o dispositivo, porque ele não esta preparado para este dispositivo, por esse motivo, existe uma divisão interna no SO. Existe uma porta especifica para comunicar com esses tipos de dispositivos. O nome em inglês para esta porta é **Device Driver**. Para dispositivos padrões já existem os devices drivers originais.

DEVICE DRIVER

O device driver é um pedaço do Sistema Operacional que controla os dispositivos. Antigamente no Unix ao trocar um dispositivo tinha que trocar redigitar a linha de código do sistema operacional, hoje em dia, ficou uma coisa isolada do SO para que ao trocar o dispositivo basta trocar o device driver. Até porque hoje em dia quem programa o device drive muitas vezes não é quem desenvolve o SO, mas sim quem desenvolve o dispositivo.

Os dispositivos não padrão precisam instalar o Device Drive deste dispositivo. Existe um device drive para cada tipo de dispositivo. O device drive fornece uma visão de mais alto nível para o sistema operacional. Este módulo do SO responsável pela comunicação entre o SO e o dispositivo. Impede que ordens erradas sejam enviadas ao dispositivo. Tem os detalhes da comunicação dos dispositivos.

Dois tipos de informação a ser enviada ao dispositivo: Dado e Controle / Status. Esses tipos de informação geram dois tipos de porta: Dado e Controle / Status. As instruções E/S são executadas no modo privilegiado, isto é, garante a execução feita só pelo SO.

ENTRADA/SAÍDA MAPEADA NA MEMORIA

Existem CPUs que não tem instruções especificas de E/S, mas mesmo assim é possível fazer a comunicação entre os dispositivos. O que acontece é que existe outro mecanismo para fazer comunicação entre os processos. Esse mecanismo se chama **Entrada e Saída mapeada na memória**. Um intervalo de endereço de memoria é reservado para comunicação com os dispositivos.

Então ele envia e recebe os dados como se tivesse escrevendo dentro da memória. Digamos que o intervalo seja: INTEL - Endereço 1000 a 1999. Por exemplo, MOV [1200], BH → Equivale a um OUT e MOV AH, [1200] → Equivale a um IN.

Digamos que estivéssemos escrevendo um dado na memória. O endereço que estávamos escrevendo não existe na memória, na verdade é um OUT, estou enviando um dado para certo dispositivo. Quando escreve equivale a enviar um dado para um dispositivo (um OUT). Enviar o dado para o dispositivo, que envia para suposta memória, que não é de verdade.

O IN funciona da mesma maneira, lemos o dado dessa memória, mas na verdade estamos recebendo um dado do dispositivo. Não reflete na memória. Enviamos e recebemos de dispositivos através desse endereço de uma memória fictícia.

Então quando a maquina não tem esse tipo de instrução especifica o mecanismo é esse. Não acessa a memória de fato usa-se um endereço reservado. Na INTEL não é necessário, pois tem as instruções IN/OUT, mas tem um caso especial na INTEL que precisa usar essa simulação de endereço reservado é o caso da placa de vídeo.

2) COMPATIBILIZANDO VELOCIDADE (SINCRONISMO) ENTRE CPU E DISPOSITIVO

A CPU consegue receber e enviar dados muito rapidamente, mas não é comum um dispositivo enviar/receber nessa velocidade. A placa de rede é uma exceção e pode ser compatível com a velocidade da CPU.

Os dispositivos E/S trocam informações. Existem três tipos de informações:

1) Dado propriamente dito (Entrada e Saída)
2) Controle ("Ordem" para o dispositivo) (Saída)
3) Status do dispositivo (Entrada)

Dado Propriamente Dito é aquele que vai ser enviado e recebido. No caso de IN o dado, por exemplo, é o que foi digitado no teclado e no caso de OUT o dado, por exemplo, é a cor do pixel que será exibido na tela.

Controle, por exemplo, a posição em uma placa de vídeo, ao solicitar a escrita de algo no disco precisa se dizer onde vai ser escrito (localidade). Outro exemplo, placas de vídeos de diferentes resoluções, é algo que pode mudar, é uma informação de controle.

Status do dispositivo, por exemplo, uma impressora precisa de papel para imprimir, mas não tem papel na bandeja. Então, ela avisa que não tem papel. Para que isto aconteça existe um retorno da impressora que informa que ela não tem papel, não é um dado é uma informação no caso de erro.

O **Dado** é enviado e recebido o **Controle** e o **Status** não. Na INTEL o Controle é sempre OUT e o Status é sempre de entrada IN.

É preciso estabelecer as portas dos dispositivos para se comunicar com a CPU. Então temos dois tipos de porta: ***porta de dados*** e ***porta de controle status***. A porta de dados envia/recebe dados, mas o uso é diferente, pois depende se é IN ou OUT.

Precisa ter um mecanismo que regulariza o sincronismo na transferência de dados, pois a CPU é muito mais rápida que o dispositivo então tem que compatibilizar a velocidade. Existem algumas formas de fazer essa compatibilização:

- **Espera Ocupada (ou Polling)**
- **Interrupção**
- **DMA**

ESPERA OCUPADA OU POLLING

Antes de enviar o dado a CPU verifica se o dispositivo pode receber. A CPU executa o loop e espera que o dispositivo venha estar disponível. Ela sabe se o dispositivo está disponível através de uma porta de status. A CPU consulta essa porta de status e verifica se o dispositivo está disponível. Enquanto a CPU está esperando o dado ficar pronto no dispositivo, ela fica em loop ociosamente, só executando duas instruções uma de IN e outra de OUT. Não executa mais nada. Existem outras formas de E/S mais eficientes de sincronizar a velocidade. A seguir um exemplo em linguagem de máquina de Espera Ocupada.

```
LOOP :  IN BH, 201 // 201 é a Porta de Status
        CMP BH, 0
        JNZ LOOP
        OUT 200, AH // 200 é a Porta de Dados
```

De tempo em tempo, ou o tempo todo, a CPU tem que monitorar o status do dispositivo. Fica perguntando no caso de IN: Você tem o dado pronto? Você tem? Se não tiver, a CPU tem que esperar o dado. Se já tiver o dado faz o IN.

O IN no registrador qualquer e a porta do status. Dependendo do valor do status sabe se o valor está disponível ou não. Se for zero não tem o dado ainda. Então pula para trás se não for zero o dado é enviado. Enquanto for zero ele fica no LOOP. Quando tiver o dado disponível ele responde com outro valor na porta de status e sai do LOOP e faz o IN na porta de dados recebendo dados do dispositivo.

Apesar de se gastar um tempo relevante da CPU para monitorar o dispositivo os outros processos continuam executando devido o time line, mas é um desperdício de capacidade da CPU.

Em alguns casos isso não é problema, no caso antigo, a maquina rodada um processo por vez, então não tinha problema. O processo que esta enviando o dado está esperando esse dado ficar pronto, mas como só existe esse processo, não tem

problema. Mas, em uma maquina multiprocessador isso é um problema, pois a CPU poderia estar executando algo enquanto o dado não está pronto. Esse mecanismo não é muito utilizado hoje em dia.

Vantagens da Espera Ocupada

- Garantia de dado válido;

Desvantagens da Espera Ocupada

- Desperdício da CP, pois poderia executar um código útil;

INTERRUPÇÃO

O dispositivo avisa quando está pronto ao invés da CPU perguntar se está pronto. O aviso é feito através do mecanismo de hardware chamado Interrupção. A CPU pode estar executando qualquer coisa que será interrompida quando acontecer um evento.

Quando o dado fica disponível no dispositivo é gerada uma interrupção. A CPU para o que estava fazendo e executa a rotina de tratamento de interrupção e volta a executar o que estava executando antes da interrupção. No tratamento da interrupção salva se as rotinas nos registradores antes da interrupção acontecer para que continue executando de onde parou ao terminar a interrupção, mas antes de terminar tem que restaurar os registradores.

Rotina tratadora da Interrupção

```
PUSH AH
MOV AH, [300]
OUT 200, AH
POP AH
IRET
```

Rotina tratadora da Interrupção

```
PUSH AH
PUSH EEX
PUSH ELX
MOV EBX, 300
MOV ELX, 512
LOOP: MOV AH, [EBX]
        OUT 200, AH
        IN EBX
        DEC ELX
CALL Desbloqueia_Processo
POP ELX
POP EBX
POP AH
IRET
```

O dispositivo gera uma interrupção. Nesse caso a CPU não fica monitorando o status do dispositivo. Ele é que muda o seu status quando fica pronto e gera uma interrupção. A CPU para de fazer o que estar fazendo e executa a rotina que trata essa interrupção.

Vantagens da Interrupção

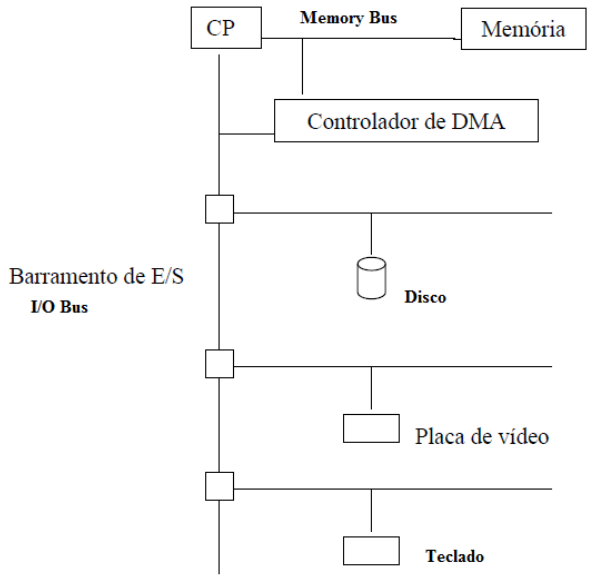
- A CPU não fica ociosa perguntando se o dispositivo está pronto, pois ele avisa que o dado está pronto;

Desvantagens da Interrupção

-

POR DMA (DIRECT MEMORY ACCESS)

DMA é um acesso direto de memória. É uma comunicação entre a CPU e a memória que é feita por um componente chamado barramento de memória que sai da CPU e vai para memória. Existe outro barramento que é o de E/S. Se terminar de usar um dispositivo de E/S, o dado tem que estar na memória. No caso de fazer E/S com DMA existe um componente que se chama controlador de DMA, este se liga no barramento de E/S e no barramento de memória, a função deste controlador é ser capaz de fazer o dado transitar entre o dispositivo de E/S e a memória sem ter que passar pela CPU.



O Disco e a Placa de Rede utilizam a DMA porque o volume de dados é alto.

Execução Normal do SO:

Bloquear o processo que motiva a leitura
Ordem para o dispositivo
Ordem para o controlador de DMA
Escalonar outro processo

O SO dá a ordem para o dispositivo ler e dá ordem para o DMA transferir e escalona outro processo para executar. Após a execução normal da CPU num certo momento todo dado que o processo quer estará disponível na memória. Quando o dado for transferido para memória o processo é desbloqueado, pois o dado que ele queria está na memória.

O Barramento de Memoria é um conjunto de fios. OUT: Nº da porta + o dado → Só quem tem a porta que usa.
IN: Recebe o dado do dispositivo para o registrador. Hoje em dia não se usa interrupção para colocar os dados no disco.

O SO sabe que o dado está disponível para o processo usar através de um tratador de interrupção que tem como objetivo avisar que o dado está na memória. Antes a interrupção transferia o dado para a memória, agora só avisa que o dado está na memória no final. A transferência é feita pelo controlador de DMA. O SO informa para controladora de DMA: Qual a porta, a quantidade de bytes e o endereço.

Portanto, todos os dispositivos que precisam compatibilizar velocidade de transferência de dados com a CPU e cujo volume de dados é muito grande utiliza esse mecanismo, o DMA, que é o mais eficiente. Quando o volume de dados é pequeno é utilizada a interrupção. Exemplos de dispositivos que utiliza interrupção transferem 1 byte por vez: mouse, teclado, modem.

Vantagens da DMA

- A vantagem é que a CPU não gasta tempo possuindo o dado, lendo ou escrevendo no dispositivo, pois o controlador que faz isso;
- Torna mais rápida E/S;
- Vantajoso quando tem grande volume de informação, pois enquanto a controladora de DMA faz transferência, a CPU está fazendo outras coisas. Por exemplo, um dispositivo quer ler um arquivo, bloqueia o processo, quando o arquivo tiver sido lido, o processo é desbloqueado. Com DMA só existe um bloqueio sem DMA tem dois bloqueios. Faz sentido fazer DMA quando o volume é alto, pois compensa o tempo gasto no DMA, se for pequeno não compensa.

Desvantagens da DMA

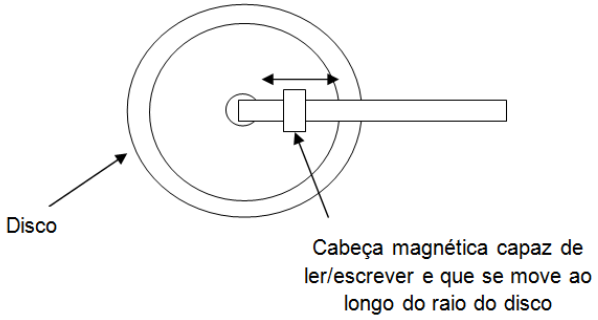
- Concorrência no uso do barramento, pois a CPU e a Controladora de DMA podem querer usar o barramento no mesmo instante e a CPU poderá ter que esperar a controladora de DMA largar o barramento. Apesar de ser uma perda de tempo a CPU ter que às vezes esperar a desocupação do barramento, essa perda é muito pequena comparada com o tempo gasto na transferência de dados por interrupção.

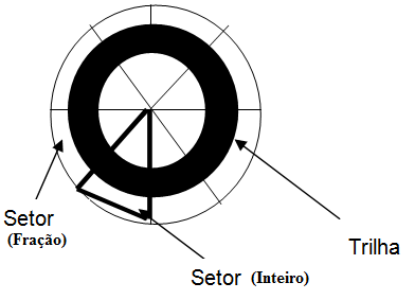
Observações:

- Não se sabe se há prioridade para uso do Controlador de DMA ou da CPU. Existe um chip, que é o árbitro do barramento, onde pode configurar quem utilizará o barramento, a CPU ou o DMA.
- A DMA não resolve todo problema de IN/OUT, pois o fato do dado está na memoria não significa que ele pode rodar. Ainda falta a etapa de desbloquear o processo. O DMA gera uma interrupção para avisar ao SO que a transferência acabou e ele desbloquear o processo.
- Por DMA é mais eficiente quando o volume de dados é alto. Quando é baixa a interrupção simples resolve.
- Quem transfere os dados é a Controladora de DMA a CPU não se envolve na transferência de dados. Ele lê o dado e faz a transferência a CPU não fica ocupada nessa transferência.

3) <u>DISCO (OU DISPOSITIVO DE ARMAZENAMENTO)</u>

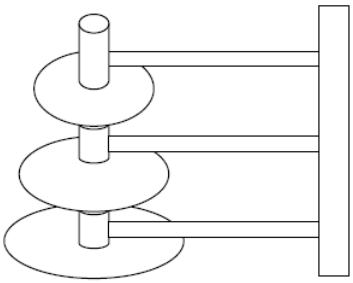
O disco tem que ser organizado de alguma forma para localizar um dado de forma eficiente. Ele é dividido em trilhas e setores. As trilhas se dividem em setores.





Cada Trilha e setor tem um numero. A Fração é a intersecao da Trilha com o Setor (Inteiro).

Hoje em dia, a cabeça magnetica funciona igual a cabeça de leitura de um disco de vinil. Quando o setor passa em baixo da cabeca a leitura acontece. Nos discos antigos era preciso identificar qual a trilha e o setor que precisa ser acessado. Os discos mais modernos têm vários pratos, um em cima do outro, com um espaçamento entre eles. Um prato tem duas superfícies. Logo, precisa ser informado três informações para gravar/ler/apagar um dado: N° da superfície, N° da trilha e N° do setor, por exemplo, Superfície 0, Trilha 6 e Setor 3.



Conforme mostra a Figura ao lado, cada superfície tem uma cabeça magnética. Existe um motor só para todas as cabeças. Logo, se uma cabeça se mover as demais se mexem também. Por exemplo, se uma cabeça se posiciona sobre a trilha 3 todas as outras cabeças magnéticas das outras superfícies também se posicionaram na trilha 3 cada uma na sua superfície.

Isso influencia no uso do disco, pois para ler um setor:
1º) A cabeça tem que se deslocar para a direção da trilha mexendo se por inteiro (***Tempo de seek)***;
2º) O setor precisa passar embaixo da cabeça através do giro do disco (***Tempo de latência)***.

Existem três tempos que somados dão o tempo total gasto para se obter um dado do Disco que são:

- **Tempo de seek** → Tempo gasto no movimento da cabeça ao longo do eixo para chegar à direção da trilha, ou seja, tempo da cabeça alcançar a trilha correta. É obtido através de teste.
- **Tempo de latência** → Tempo gasto para o setor passar embaixo da cabeça. Melhor caso: 0s. Pior caso: Tempo de 1 volta. Médio: (0 + Tempo de 1 volta) / 2.
- **Tempo de transferência** → Tempo gasto na leitura de todo um setor pela cabeça para que o dado possa ser transferido para a memória da controladora, ou seja, é o tempo da cabeça percorrer o setor inteiro. Tempo para percorrer n setores = Tempo de 1 volta. Tempo para percorrer 1 setor = Tempo de 1 volta / n de setores. Pior caso: 1 volta.

Tempo de Acesso = Tempo de Seek + Tempo de Latência + Tempo de Transferência

Geralmente o **Tempo de Seek** é de 10 ms. O **Tempo De Latência** depende do número de rotações do disco. Se o disco faz 12000 rpm (rotações por minuto), ele faz 200 rotações por segundo ou 0,2 rotação por milésimo de segundo. Portanto, para dar uma volta, o disco gasta 5 ms. Para dar meia volta, que é a distância média que o setor leva pra passar embaixo da cabeça magnética, leva-se 2,5 ms. O **Tempo De Transferência** é bem pequeno.

O tempo que demora para cabeça chegar onde você quer ler é o **Tempo De Latencia**, isso dps que a cabeça chegou na trilha correta. Ainda tem o tempo que a cabeça leva para percorrer os vetores, esse tempo é chamado de **Tempo De Transferência**.

Porque durante esse tempo que a cabeça passa por cima do vetor, o dado magnetivo gravado no disco magnético é convertido para digital e gravado na memoria interna do disco, o dado é transferido do disco magnético para controladora interna no disco. Então o somatório desses 3 tempos é chamado de **Tempo De Acesso**. Então dps que a transferência acontece o dado está contido na memoria interna do disco, o passo ainda não está completo, pois ainda precisa passar

para a memoria principal (é a DMA que faz isso!).

Digamos que temos 50 setores por trilha. Tempo de transferência é o tempo que demora para percorrer um setor, 50 setores é igual uma volta inteira. Qual o tempo para percorrer um setor? Se 50 é uma volta, então o tempo para percorrer um setor é igual a Tempo de uma volta/50.

Digamos que tenhamos um arquivo que ocupe 4 setores, ele pode ficar em 4 setores distintos. A vantagem de colocar separado é porque é fácil de alocar enquanto tiver espaço em disco, o problema vai ser na hora de ler, pois terá latências, diferentes. Por outro lado, se tivermos um arquivo em sequencia à latência vai ser menor. Já que gastará apenas um tempo de latência.

A estrutura do disco em forma de "Pizza" tem uma deficiência, pois gera um desperdício. O setor interno é bem menor que o externo, ou seja, a quantidade de bytes é menor. A capacidade de magnetização é proporcional à massa. Pode-se guardar 1kb em ambos, mas no setor maior pode-se guardar 4 kb em vez de 1 kb. Ai está o desperdício da capacidade de magnetização, pois pode guardar mais.

Atualmente os discos não tem mais essa estrutura. O numero de setores por trilhar é constante, nas trilhas mais internas tem menos setores, conforme vai aumentando o numero de trilhas vai tendo mais setores. Na primeira tem 4, na segunda tem 5, na outra tem 6 e assim por diante, a ideia é que desse jeito o comprimento seja mais ou menos igual, praticamente constante tendo menos desperdício.

O setor mais interno tem um movimento menor que o setor mais externo. Existe uma capacidade de armazenamento maior no setor mais externo. A ideia é gravar a mesma quantidade de byte no setor mais externo e no mais interno Atualmente nas trilhas mais internas existem menos setores, enquanto que nas trilhas mais externas tem mais setores.

Capacidade de magnetização é proporcional a Massa do Material
Capacidade de magnetização é proporcional a Densidade*Volume
Capacidade de magnetização é proporcional ao Volume
Capacidade de magnetização é proporcional a Profundidade*Largura*Comprimento
Capacidade de magnetização é proporcional ao Comprimento

Onde: Densidade, Profundidade e Largura são constantes.

Desperdício ao dimensionar para o pior caso, por exemplo, ao colocarmos um arquivo de 1 KB em um setor que cabe 4 KB temos um desperdício de 3 KB. Para evitar esse desperdício nos discos atuais os setores internos tem aproximadamente o mesmo tamanho em todas as trilhas do disco.

Exemplo: C1 = 1/3 (Tempo) e C3 = 1/7 (Tempo). Nesse caso o tempo é sempre igual tanto no setor interno quanto o setor externo.

Exemplos de Dispositivos que armazenam dado: CD ROM / Disco rígido / Disco Flexível. E Dispositivos que não armazenam dado: Placa de Rede/Teclado/Mouse/Modem.

Os dispositivos de não armazenamento de dados são chamados de dispositivos de caractere, pois não há a necessidade de ter a unidade localizável. O Windows usa esse tipo de dispositivo indiretamente. Tem um conjunto de rotinas do SO para lidar com cada tipo de dispositivo (teclado, mouse e etc.) são chamadas que o programador pode fazer. Cada dispositivo que é inventado usará uma API que já existe ou terá que inventar uma API nova.

O Unix tem uma maneira diferente na chamada ao dispositivo. O dispositivo é como um arquivo os dados podem ser mandados aos dispositivos como se estivesse escrevendo ou lendo em um arquivo, isso diferencia a programação.

Todo setor tem no final informação de controle para permitir a verificação da integridade do dado que foi gravado no setor. É vantajoso salvar um arquivo em vários setores da mesma trilha. Tempo de seek e latência menor (não se gasta). Gasta apenas o de transferência. Quando a trilha acaba o melhor próximo setor é na superfície de baixo ou em outros pratos. De um cilindro para outro gasta tempo de seek.

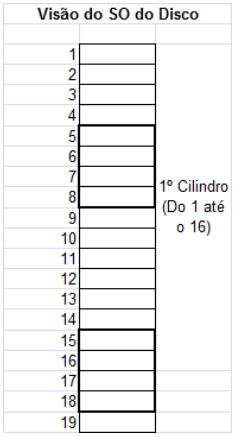
Hoje em dia, o disco é visto como um vetor de setores. Cada cilindro ocupa setores em sequencial. O SO guarda um arquivo de forma sequencial em setores sequenciais. No melhor caso, só será gasto Tempo de seek e Tempo de latência inicial.

LINEARIZAÇÃO DO DISCO (OU DISCO LÓGICO)

O SO vê o disco como unidimensional. Essa linearização é feita pelo device driver do HD. Quais os tipos de acesso que podem ser feitos nos arquivos? Sequencial e Randômico.

Acesso sequencial: o arquivo é lido em sequência por um programa (compilador ou acesso ao banco de dados de empregados cuja pesquisa é feita do início do disco).

Acesso randômico: o arquivo é lido em uma ordem qualquer (acesso ao banco de dados de empregados com pesquisa através de índice que localiza diretamente o empregado).



O device driver é que faz a arrumação do arquivo na mesma trilha em várias superfícies, a fim de agilizar o acesso a um arquivo. O cilindro é o conjunto das mesmas trilhas em superfícies diferentes. Só se usa outro cilindro se a mesma trilha de todas as superfícies já foram usadas.

Quando existem diferentes processos querendo ler dados do disco, os pedidos precisam ser coordenados para que possam ser atendidos. A ideia é fazer com que a cabeça magnética mexa o mínimo possível. Esta é uma visão que é complicada para o SO, pois ele não conhece as três dimensões. Então existe uma função no hardware que se chama **linearização do disco (ou Disco Lógico)** que transforma o disco em uma coisa lógica com uma dimensão. O hardware mostra para o SO uma visão do disco como uma visão linear.

Observações:

- Para o acesso sequencial o arquivo precisa ser colocado na mesma trilha. Se o arquivo for maior, o ideal é que o resto do arquivo seja colocado na mesma trilha, mas em superfícies diferentes, já que todas as cabeças se movimentam juntas e com esse esquema o dado é lido mais rápido (reduz o tempo de seek).

- Desfragmentação só faz sentido em Disco Magnético.

ALGORITMOS DE ESCALONAMENTO EM DISCO

Digamos que no instante zero a cabeça esteja na trilha 20. Se ninguém usa o disco a cabeça não se mexe. Ela continua na trilha 20 e o disco continua rodando. Digamos que nesse instante de tempo existe um pedido e esteja no cilindro 30. Então a ordem é enviada para o disco e ele começa a mexer até alcançar o lugar onde tem que chegar e a cabeça vai se mexer até chegar onde pretende chegar. Chegando nesse lugar a cabeça vai ler e depois se mexe até o próximo requerimento.

Enquanto isso outros processos estão executando e esse processo que fez o pedido está bloqueado. A questão é que se mais de um processo fizer pedido de leitura ao disco Qual vai ser atendido primeiro?

Digamos que tenhamos um pedido do cilindro 70 e outro pedido do cilindro 10. Então temos 2 pedidos pendentes: 10 e 70, se formos fazer por ordem chegar, primeiro iremos no 70 e depois o 10, sai do 0 e vai pro 70 e depois vai para o 10. Enquanto estar no 70, pode chegar outro pedido, digamos que chegue o pedido do cilindro 60 e 25. Então depois do 70 vai para o 10 e depois vai parar o 25.

Fazendo na ordem de chegada existe um risco de ter um desperdício, pois percorre caminhos longos varias vezes, melhor do que isso seria fazer primeiro os pedidos mais próximos em vez de fazer zig-zag. Mas, nada impede de sempre chegar novos pedidos, e pode ocorrer de alguns ficarem muito tempo esperando isso não é muito justo. Essa logica gera um tempo total mais curto, mas não é justa. Esse algoritmo não é usado. O outro não é bom. Esses dois são casos de escalonamento de disco são Algoritmos de escalonamento.

O primeiro apresentado é chamado de FCFS e o segundo é SSTF.

Existe uma terceira logica que introduz um conceito de pedido e movimentação, que pode ser positivo ou negativo, se a cabeça está indo para números maiores, sentido positivo, se for para sentido menos, sentido negativo. É o elevador.

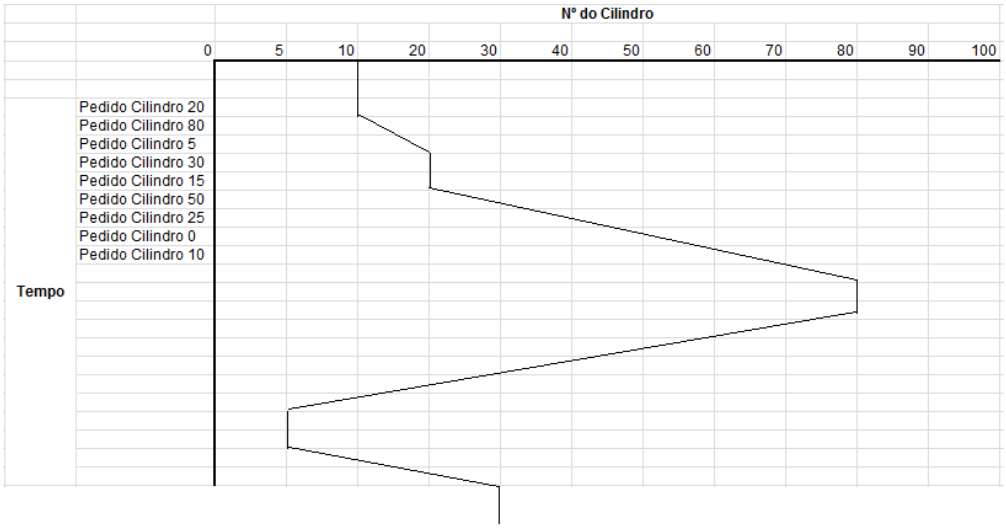
Apesar de o cilindro estar mais próximo o algoritmo do elevador vai escolher o que estiver no sentido atual. Se não tiver mais ninguém nesse sentido "positivo", então muda o sentido do elevador, ou seja, começa atender no sentido "negativo". Apesar de dar um resultado pior que o SSTF não foi injusto e foi melhor que o FCFS, pois um pedido distante não fica muito tempo sendo postergado é atendido com a justiça devida. O único problema é que privilegia os pedidos do meio.

Tem uma modificação que não privilegia pedidos do meio o nome desse algoritmo é o Elevador circular. Ele atende sempre no mesmo sentido, ou seja, "positivo" ou "negativo".

O SO recebe o pedido transforma no vetor e manda para o disco e o disco que controla através do hw controlador de disco que gerencia o escalonamento.

FCFS (First Come First Served)

Nesse algoritmo o primeiro a chegar é o primeiro a ser atendido.



Vantagens do FCFS

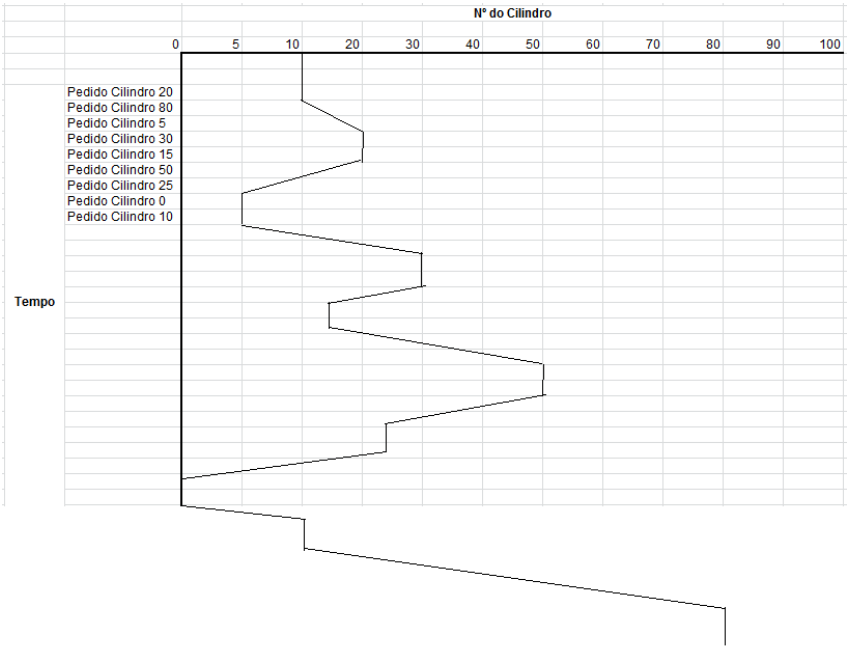
- É mais justo que o SSTF;
- É simples de ser implementado;

Desvantagens do FCFS

- Pode gerar ziguezague, pois o padrão de busco é aleatório;
- É muito lento;

O SSTF (Shortest Seek Time)

O algoritmo SSTF seleciona o pedido com o tempo de busca mínimo a partir da posição atual da cabeça. Como o tempo de busca aumenta com o número de cilindros percorridos pela cabeça, o SSTF escolhe o pedido pendente mais próximo da posição atual da cabeça. Ele é basicamente uma forma de escalonamento Shortest Job First (SJF).



Vantagens do SSTF

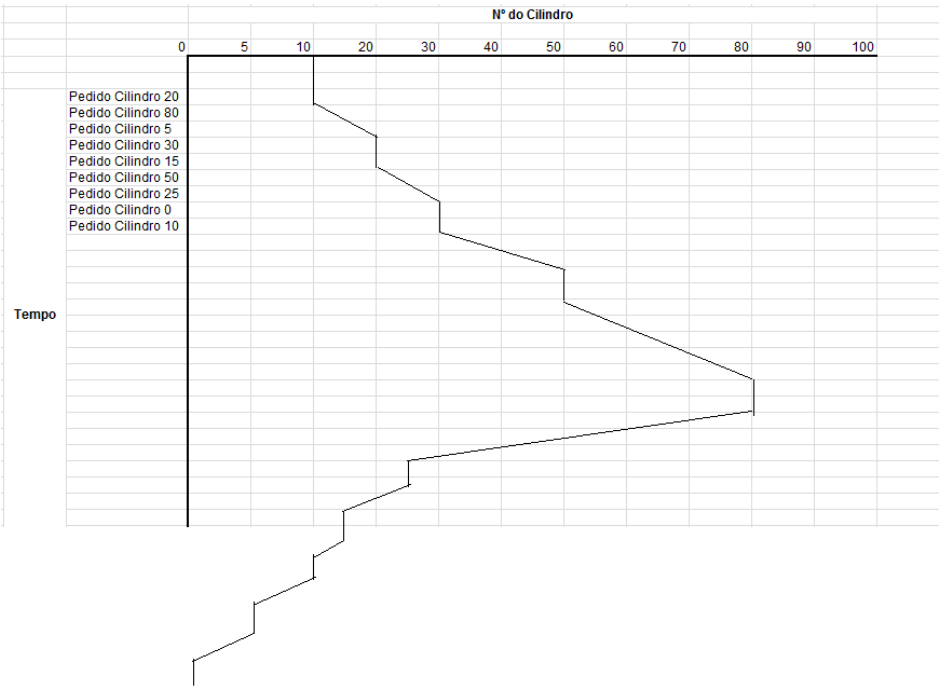
- Evita o ziguezague, pois reduz o movimento da cabeça;
- É melhor do que o FCFS;
- O Tempo de Seek é menor.

Desvantagens do SSTF

- Os pedidos mais distantes sofrem Estarvation (paralização), pois um cilindro pode estar distante dos demais;

Elevador (ou SCAN)

No algoritmo do Elevador a cabeça do disco começa em uma ponta do disco e se movimenta em direção à outra ponta atendendo aos pedidos assim que chega a cada cilindro até atingir a outra ponta do disco. Quando atinge a outra ponta do disco o sentido do movimento da cabeça é invertido percorrendo o disco no outro sentido. Ao chegar à ponta e voltar, no início dessa volta tem poucos pedidos, pois a cabeça do disco acabou de passar por ali, ficando um maior número de pedidos e mais velhos na outra ponta. Dessa forma os pedidos do meio serão privilegiados.



Vantagens do Elevador

- É mais justo que o SSTF, pois atende aos pedidos de acesso a cilindros mais distantes;
- É mais rápido do que o Elevador Circular;

Desvantagens do Elevador

- Os pedidos do meio são privilegiados;
- É menos eficiente do que o SSTF;

Elevador Circular (ou C-SCAN)

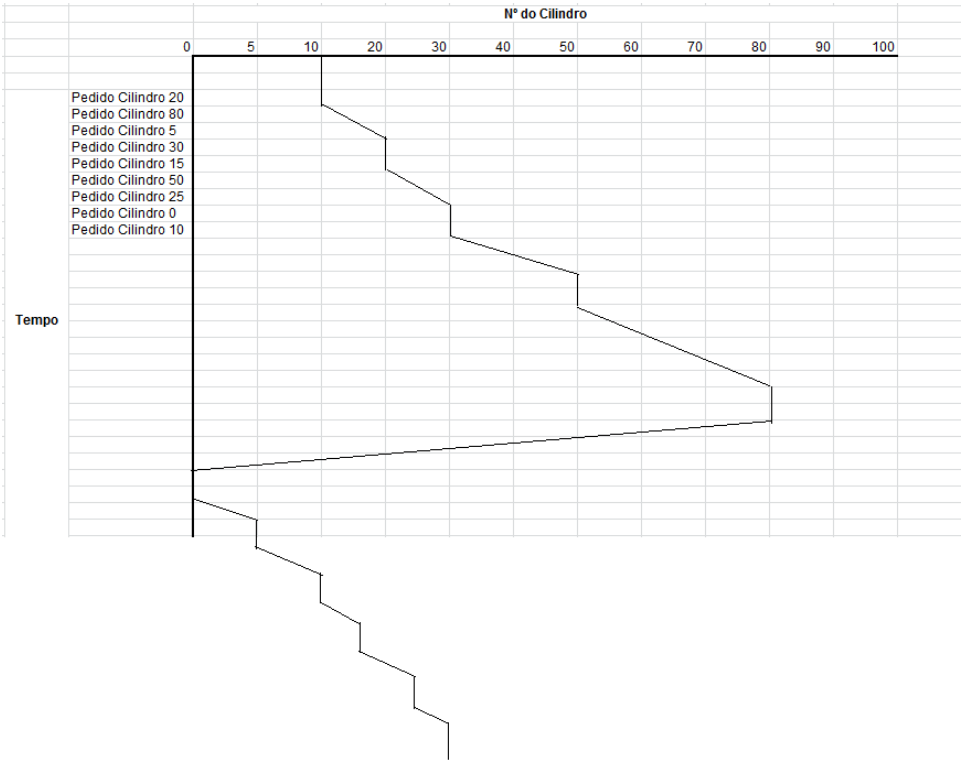
O algoritmo Elevador Circular é uma variante do Elevador que foi projetado para fornecer um tempo de espera mais uniforme. A diferença é que ao chegar à ponta do disco, ele move a cabeça imediatamente para o início da outra ponta sem atender nenhum pedido nesse trajeto. Só então percorre o disco atendendo aos pedidos, ou seja, os pedidos só são atendidos em um sentido. O nome circular é devido a este comportamento, ou seja, trata os cilindros como uma lista circular, ligando o último ao primeiro.

Vantagens do Elevador Circular

- Não privilegia os cilindros do meio;

Desvantagens do Elevador Circular

-



LOOK

Os Algoritmos Elevador (SCAN) e Elevador Circular (C-SCAN) são chamados de LOOK e C-LOOK, porque procuram por um pedido antes de continuar a mover em determinado sentido.

AVALIAÇÃO DOS ALGORITMOS DE ESCALONAMENTO EM DISCO

Ao avaliar um algoritmo deste estamos avaliando velocidade e justiça. Dentro todos os algoritmos citados acima o SSTF é o mais rápido de todos.

Quem roda o algoritmo é o disco, o SCSI, que aceita vários pedidos, pois o IDE aceita apenas um pedido por vez e trava o barramento quando está atendendo e quem envia o pedido é o Sistema Operacional. Normalmente é o Elevador o algoritmo utilizado.

Na seleção de um algoritmo de escalonamento de disco o SSTF é comum e tem um grande apelo, mas o Elevador e o Elevador Circular têm desempenho melhor em sistemas com uso pesado de disco, porque têm menos probabilidade de ter o problema de paralisação. Com qualquer algoritmo de escalonamento, o desempenho depende muito do número e dos tipos de pedidos.

Por exemplo, se a fila tiver um único pedido em espera, todos os algoritmos de escalonamento terão o mesmo comportamento. Todos se comportarão como o escalonamento FCFS. O método de alocação de arquivo também influencia os pedidos de acesso a disco. A leitura de um arquivo alocado continuamente gerará vários pedidos próximos no disco, resultando em um movimento limitado da cabeça. Por outro lado, um arquivo indexado ou encadeado pode estar "espalhado" no disco provocando muita movimentação da cabeça.

A localização da estrutura de diretório e blocos de índices deve estar o mais próximo possível dos dados evitando muita movimentação da cabeça entre a pesquisa e o acesso aos dados. Armazenar em cache na memória principal os diretórios e blocos de índice também pode ajudar a reduzir o movimento da cabeça para pedidos de leitura.

Os algoritmos descritos acima estão levando em conta a distância de busca, mas um fator que também influencia muito é a latência rotacional. Os discos atuais não revelam a posição física dos blocos, logo fica difícil para o Sistema Operacional escalonar para melhorar a latência rotacional. Para resolver esse "problema", os fabricantes de disco incorporam no hardware de disco, hardware de controle com implementação de algoritmos de escalonamento de disco.

Neste caso, o Sistema Operacional pode enviar um lote de pedido para a controladora, ela pode colocar na fila para escalonar e melhorar o tempo de busca e a latência rotacional. O hardware de disco melhora o desempenho de I/O. Porém só isso não basta, então entra o Sistema Operacional para controlar a ordem de serviço de pedidos.

O Sistema Operacional pode escalonar os pedidos e enviar um a um para a controladora de disco, de acordo com a prioridade. Por exemplo, se o cache estiver com poucas páginas livres, ele pode dá preferência à escrita (na escolha de

página vítima) em disco e não a leitura. Outro exemplo pode ser o temporizador, que de 30 em 30 segundos no UNIX e poucos segundos no Windows grava as informações dos blocos modificados da cache em disco.

O Sistema Operacional também é responsável pela inicialização dos discos, a carga do sistema a partir do disco e a recuperação de blocos defeituosos.

4) ARQUIVOS

Arquivo é qualquer informação que esta salva em disco. O arquivo contém informações de controle, por exemplo, que horas foi alterado, qual foi o usuário que o criou, onde está no disco, etc. É o SO que guarda estas informações sobre o arquivo.

O arquivo serve para preservar uma informação quando a máquina é desligada, pois a memoria perde todos os dados quando a máquina é desligada. Os arquivos são armazenados no Disco. Os arquivos têm Nome, Conteúdo e Informações de Controle (também conhecido como Metadados, mas é menos utilizado).

Qualquer SO precisa de no mínimo o Nome é o Conteúdo para salvar um arquivo. A seguir é apresentado o que existe em cada um:

Nome

Normalmente é uma string que contem letras e/ou números. Existem variações que podem ocorrer de um SO para outro: O tamanho do nome do arquivo, pois existe um tamanho máximo e o formato Maiúsculo e Minúsculo que pode ser:

- Só Maiuscula. Ex.: MS DOS e Z/OS (Mainframe IBM)
- Maiuscula e Minuscula sem diferenciação, ou seja, 'a' = 'A'. Ex.: Windows
- Maiuscula e Minuscula com diferenciação, ou seja, 'a' != 'A'. Ex.: Unix

Tamanho máximo: MS-DOS = 11 caracteres, Z/OS (Mainframe IBM) = 40 caracteres, UNIX e WINDOWS = Limites grandes.

Conteúdo

É o nível de conhecimento que o SO tem sobre o conteúdo do arquivo. Cada SO trata o conteúdo do arquivo de uma forma diferente. Existem três tipos de Conteúdo:

A) Uma sequencia de registros. Ex.: Z/OS (Mainframe IBM)

Registros de tamanho fixo ou variável. Ex.: Z/OS Facilita a vida do programador. Não é típico de SO. Hoje em dia não é assim que o SO funciona é um recurso de banco de dados, mas na época a IBM incorporou no SO para facilitar a vida do programador. O SO ficou mais complicado.

A unidade de leitura desse arquivo obriga o programador a ler um registro por vez, não dá para ler um pedaço do registro, a ideia dessa concepção, é ajudar o programador a organizar o arquivo.
Isso é interessante num banco de dados, no caso um registro ter a chave, a partir da matricula você pode ter o registro do banco de dados desse empregado. Dado com campo definido como chave, mais tarde pode-se pedir pra ler o registro com essa chave, e o SO retorna o registro com essa chave. Antigamente a IBM incorporou essa capacidade, atualmente o SO não faz isso, pois o SGBD já faz.

A unidade de leitura que o programador tem para ler e/ou escrever nesse arquivo é o registro. É obrigatório ler e salvar um conjunto inteiro de registros. Tem muito haver com a questão de guardar dados de forma organizada. Antigamente não existia SGBD (Sistemas Gerenciadores de Banco de Dados). Então os programas que precisavam armazenar dados salvavam os dados em arquivos e esses arquivos eram guardados no disco por registros.

Na época para facilitar os usuários que trabalhavam como armazenamento de dados. Então, a IBM evolui seu SO e incorporou a indexação que atualmente é típico dos bancos de dados. Isto facilita o acesso/localização dos arquivos , mas traz um complexidade para o SO tornando-o mais lento.

B) Uma sequencia de bytes. Ex.: Unix e Windows Antigo (até 98)

Nesse tipo de conteúdo o arquivo é uma estrutura simples na visão do SO que não conhece nada do que foi salvo. Ele só faz o que manda. Se mandar ler ele lê se mandar salvar ele salva. Nessa concepção a interpretação dos dados é feita pelos programadas que mandaram ler e escrever.

Na concepção de registros o SO tinha um trabalho maior, indexava os arquivos, entendia um pouco do conteúdo que estava sendo guardado nessa concepção isso não acontece. O SO que utilizamos atualmente tem o núcleo e outros programas que vem como ele. O programa mais importante que vem junto com o SO é a interface com o usuário (Interpretador de Comandos).

Hoje em dia, os interpretadores de comandos são gráficos. No caso do Windows, o nome de desse programa é o Explorer.exe é responsável pela parte gráfica do SO (interface com o usuário). Ele não pertence ao núcleo do SO é um programa externo, mas é instalado junto com o SO.

O Explorer.exe utiliza a extensão do arquivo para saber para em qual programa mandará o SO rodar para abrir o arquivo. Como o SO não sabe nada sobre o conteúdo do arquivo ele trata todos os tipos de extensão de arquivos por igual, ou seja, uma sequencia de bytes. São os programas externos que tratam as extensões como devem ser tratadas, interpretam. Com isso a complexidade sai do SO.

O programa sempre vai interpretar o arquivo mesmo que não seja de sua extensão. Ele abre como ele conhece, ou seja, se pedir para o word abrir uma imagem jpeg, ele vai abrir não dará erro porque é uma sequencia de bytes e ele traduz para o que conhece.

Ex.: Unix, Windows antigo (até o Windows 98). Versão simples do SO. Fazer de forma simples e mais rápido possível (tirar a complexidade). Núcleo do SO X Programas que são instalados junto com o SO, mas não fazem parte do SO. Ao instalar o Windows tem o núcleo do SO e os programas.

É importante diferenciar uma coisa é o núcleo do SO que implementa as chamadas de Sistemas e outra coisa são os programas que estão instalados dentro do SO. Por exemplo, digamos na instalação do Windows o núcleo do SO não entende o arquivo de um .jpg para ele é tudo byte, mas existe um software que lê .jpg. Não é o SO que interpreta é um programa especifico que lê imagem.

Por exemplo, se mandar o NOTEPAD abrir "Img.jpg". O SO deixa o NOTEPAD abrir o arquivo e o NOTEPAD interpreta como texto (as sequencias de bytes). O núcleo do SO não controla isso. Pode-se com qualquer programa abrir um arquivo que não é o tipo daquele arquivo, mas vai ser interpretado errado.

Antigamente o SO tinha noção de registro, mas hoje em dia não tem mais.

C) Várias sequencias de bytes. Ex.: Windows Novo

O SO novo da Microsoft interpreta o conteúdo dos arquivos como sendo várias sequencias de bytes. Este tipo de interpretação é como se o conteúdo do arquivo pudesse ter vários conteúdos simultâneos, ou seja, o mesmo arquivo tem vários conteúdos simultâneos e cada conteúdo é uma sequencia independente de bytes.

Isso existe no Windows Novo (NT em diante), mas não é muito usado porque os outros SOs não tratam dessa forma. Eles tratam como um conteúdo só e não vários conteúdos simultâneos e independentes.

A capacidade do arquivo ter mais de um conteúdo não é muito utilizada, pois existe somente no Windows. Exemplo de utilização desse tipo de conteúdo quando o Internet Explorer abaixa um arquivo ele é salvo em algum lugar do disco. Se daqui um ano você tentar usar esse arquivo o IE faz a seguinte pergunta "Esse arquivo já foi baixado da internet tem certeza que você que usar?".

O IE guarda no conteúdo default a informação que o arquivo foi baixado da Internet. O Explorer quando você manda usar esse arquivo ele programa no conteúdo do controle de arquivo se o arquivo foi baixado e faz a pergunta se deseja usar. Essa é uma questão de segurança.

A Microsoft chama as várias sequencia de bytes de Stream. Um Stream é diferente do outro. O conteúdo é independente do outro.

Motivação da Apple para criar essa ideia: Arquivos executáveis são arquivos que contem códigos executáveis, mas também contem coisas que estão dentro do arquivo executável e não são códigos. Os recursos do arquivo executável são tudo que não é código executável, por exemplo, imagem, efeitos, ícone etc. A Microsoft evolui a ideia para Stream

Essa funcionalidade pode ser usada em programas simples. Por exemplo, na linha de comando e digitar: Notepad Alo.txt (Default) e Notepad Alo.txt:OutraTipicamente não é utilizada essa capacidade, mas existe no Windows para que alguns programas utilizem. É como se fosse vários arquivos dentro do mesmo. Entender a capacidade de dados de controle.

NOTEPAD → Abre o arquivo ALO.txt (conteúdo Default)
NOTEPAD → Abre o arquivo ALO.txt:Z (conteúdo com o nome Z)

5) <u>DIRETÓRIOS</u>

O proposito de um Diretório é organizar arquivos para facilitar os usuários a encontra-los. Diretórios contém arquivos e são arquivos. Um Diretório é um arquivo especial que só o SO manipula o conteúdo do Diretório. Os tipos de conteúdo de um diretório são:

1) **Para cada arquivo no diretório o SO guarda:**
   - Nome do arquivo
   - Referencia a estrutura de controle deste arquivo
   - Ex.: Unix, Windows Novo (após 98)

2) **Para cada arquivo no diretório o SO guarda:**
   - Nome do arquivo
   - Dados de controle
   - Ex.: MS-DOS, Windows Antigo

3) **Para cada arquivo o SO guarda:**
   - Nome
   - Dados de controle
   - Conteúdo do arquivo
   - Ex.: Z/OS (Mainframe IBM. Conhecido como Arquivo particionado). Este é o único caso que o diretório realmente contém o arquivo.

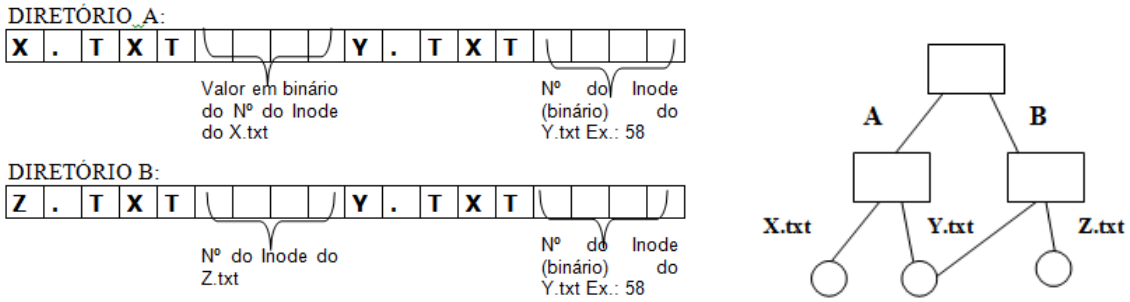
Somente o terceiro tipo que realmente guarda os arquivos propriamente ditos dentro do diretório, pois os dois primeiros apenas apontam para o local no disco onde se encontram os arquivos, ou seja, o conteúdo do arquivo não está no diretório.

No Mainframe não existe Diretório, mas sim Arquivo Particionado. Esse é o nome dado a Pasta no Mainframe. O Arquivo Particionado é interessando, pois é um arquivo que se tem vários arquivos dentro particionado. Isso lembra a um arquivo zipado.

Digamos se tenha um arquivo de 10 GB e se deseja mover esse arquivo do diretório A para o diretório B. Nos dois primeiros tipos de conteúdo de diretório a movimentação do arquivo e praticamente instantânea, pois os arquivos não precisam sair do lugar basta alterar o nome e a Referencia (ou os Dados de Controle) do arquivo. No terceiro caso fazer essa movimentação demora muito, pois o arquivo precisa ser movido.

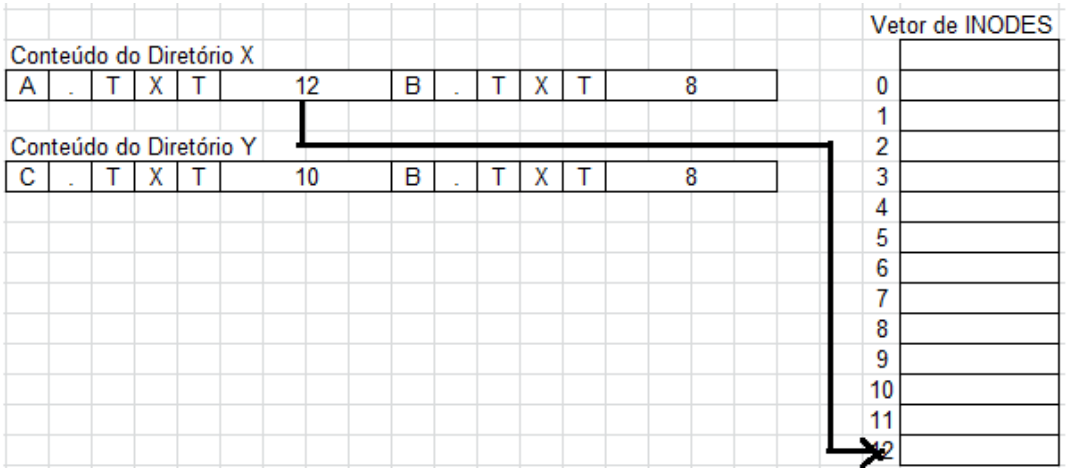
ARQUIVO EM DIRETORIOS DIFERENTES AO MESMO TEMPO

No Unix a estrutura de controle de um Arquivo é chamada de INODE. Conforme mostra a Figura abaixo.



Ambos os diretórios A e B referencia o mesmo numero de INODE (referente ao arquivo Y.txt). Assim, o mesmo arquivo se encontra em dois diretórios ao mesmo tempo. O SO sabe o que é um arquivo normal e o que é um arquivo de diretório.

O conteúdo do Diretório X teremos duas informações: uma referente ao arquivo A e outra referente ao arquivo B. O que se refere ao arquivo A tem o Nome do arquivo e a Referência aos Dados de Controle do arquivo. No Unix a estrutura que guarda os dados de controle de um arquivo é chamada de INODE. Conforme mostra a Figura abaixo:



O INODE guarda informações do tipo: data e hora de criação do arquivo, onde está salvo no disco, que tipo de arquivo é, usuário que criou, etc. O vetor de INODES fica em um lugar fixo do disco.

O conteúdo do arquivo o SO não conhece, pois para ele o conteúdo do arquivo são bytes. Diretórios são arquivos especiais e o SO continua sem conhecer o conteúdo do arquivo.

O exemplo acima está incompleto, pois o conteúdo do Diretório contem outras informações que são mais complicadas. Esse assunto não será abordado nessa disciplina. Na estrutura de controle simples apresentada que a do UNIX e do WINDOWS Novo o Diretório não tem quase nada a respeito do arquivo. Tem somente o nome e um ponteiro para os dados de controle.

O que pode ser feito nesse caso é que as informações a respeito do arquivo podem ser copiadas para outro Diretório. Logo, o Diretório B pode ter a mesma informação que do Diretório A. Basta fazer referencia ao mesmo INODE. Na prática o que temos é que o mesmo arquivo "B.TXT" está em dois Diretórios ao mesmo tempo.

Essa possibilidade de um mesmo arquivo está em dois Diretórios ao mesmo tempo só pode existir por causa da abordagem mais simplista em que o Diretório só contem a referencia aos dados de controle. Se o Diretório contem os próprios dados de controle, como é o caso do WINDOWS Velho, não tem como um arquivo estar em dois diretórios ao mesmo tempo. Pois, ao tentar fazer isso nesse SO os dados de controle serão duplicados e um arquivo não pode ter dados de controle duplicados.

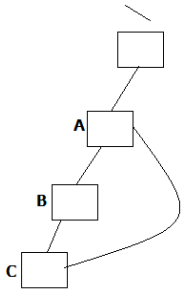
Atenção o caso de um arquivo está em dois diretórios ao mesmo tempo é diferente do caso de uma copia de arquivo. No primeiro é feita uma referencia ao mesmo dado de controle. No segundo crio um novo arquivo a partir de um arquivo existente. O primeiro caso lembra um atalho mais também não é.

A ligação de um Diretório com um arquivo é chamada de link. Quando o arquivo é criado ele tem um link, mas se pode criar outros links para este arquivo em outros Diretórios. Isso é muito comum no UNIX no WINDOWS isso existe, mas não é usado. A Microsoft não incentiva esse uso. O SO é capaz de fazer, mas ela não fornece nenhum programa junto com a instalação do Windows que faça isso. Para conseguir fazer isso no Windows é preciso baixar um programa da Microsoft que faça a chamada ao SO que o Windows tem é cria um arquivo em dois Diretórios ao mesmo tempo.

Como o proposito do Diretório é organizar arquivos. Logo, poder colocar arquivos em Diretórios diferentes quer dizer que se pode organizar os arquivos de forma diferente. Tem varias formas de se organizar arquivos.

O link é tão forte que quando um arquivo tem dois links e se apaga um dos links o outro continua funcionando normalmente. Depois que se cria o segundo link não se sabe qual dois foi criado primeiro. Um dois motivos da Microsoft não divulgar isso é porque não é fácil explicar. É mais fácil explicar que um arquivo esta em um Diretório sempre.

O Diretório referencia o INODE, não é o INODE que aponta para o Diretório. Logo, se tiver dois links para um mesmo INODE se apagar um o outro continua funcionando. O arquivo B está no Diretório X e Y ao mesmo tempo. Mas um não sabe da existência do outro somente o INODE que contem o numero de links de um arquivo. Não tem nenhum dado de controle no INODE que aponta para os diretórios que contem o arquivo.



Enquanto um arquivo tiver um link ativo o arquivo não é apagado. Somente quando o contador de links do arquivo no INODE fica zero é que o arquivo é apagado. Esse é um dos motivos da Microsoft não divulgar, pois é difícil explicar e um usuário leigo entender que o arquivo sigiloso ainda não foi apagado realmente, pois existe outra referencia ao arquivo em algum lugar do sistema.

O link tem essas complicações de apagamento de backup e tem um caso que é mais complicado ainda que pode gerar um loop infinito conforme mostrado na Figura ao lado.

Observações:

- A mudança dos arquivos de diretório nos tipos de conteúdo do diretório 1 e 2 é praticamente instantânea, pois é só mudar à referência. Porém, no tipo 3 é demorada.

LINK (OU HARD LINK)

São cópias de uma entrada do sistema de arquivos. As duas entradas contém nomes diferentes, mas apontam para o mesmo local físico no disco (INODE). Compartilhando, portanto, além do mesmo conteúdo as mesmas permissões. Se o

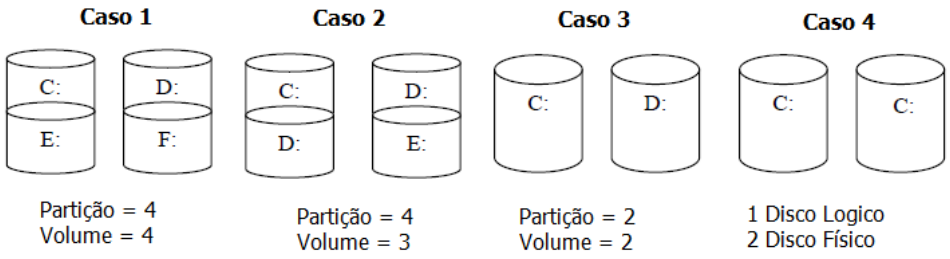
arquivo verdadeiro for apagado, o hard link continua apontando para o mesmo local físico sendo, portanto, acessível da mesma forma.

O arquivo e o hard link que aponta para ele devem obrigatoriamente estar localizados no mesmo sistema de arquivos já que o hard link aponta para um endereço físico (INODE) e não se pode garantir que estes endereços sejam únicos em vários sistemas de arquivos, ou seja, em volumes diferentes.

Resumindo é quando o mesmo arquivo está em dois diretórios diferentes.

DISCO FISICO

Partição → Pedaço do Disco Físico.
Volume → Aquilo que é visto como disco pelo usuário. Também chamado de Disco Lógico.



No passado remoto todo disco físico era igual ao volume. Por exemplo, a maquina tinha 3 discos físicos e o usuário via três volumes. Depois surgiu a idea de quebrar o disco em partições. Partição e Volume não são a mesma coisa.

O Windows vê o disco como letras conforme mostra o Caso 1 apresentado na Figura acima. Mas, o disco pode ser apresentado como no Caso 2, na visão do usuário existem apenas 3 disco, mas na verdade são 2 discos físicos com 4 partições e 3 volumes.

Assim se consegue ampliar a capacidade de um disco sem modificar muita coisa. Basta incluir no HW que o SO vai reconhecer como continuação do disco C. Ambas as soluções tem como objetivo tornar a utilização mais fácil para o usuário. Ele não se preocupa com os volumes, pois estão em uma arvore só.

Problema: Como o usuário utiliza arquivos que estão em diferentes volumes?
Solucoes: 1) Nomeando o volume explicitamente. Ex.: Windows
2) Sem nomear o volume explicitamente. Ex.: Unix e Z/OS (Mainframe). Para ter acesso a outro volume tem que fazer montagem (usa link).

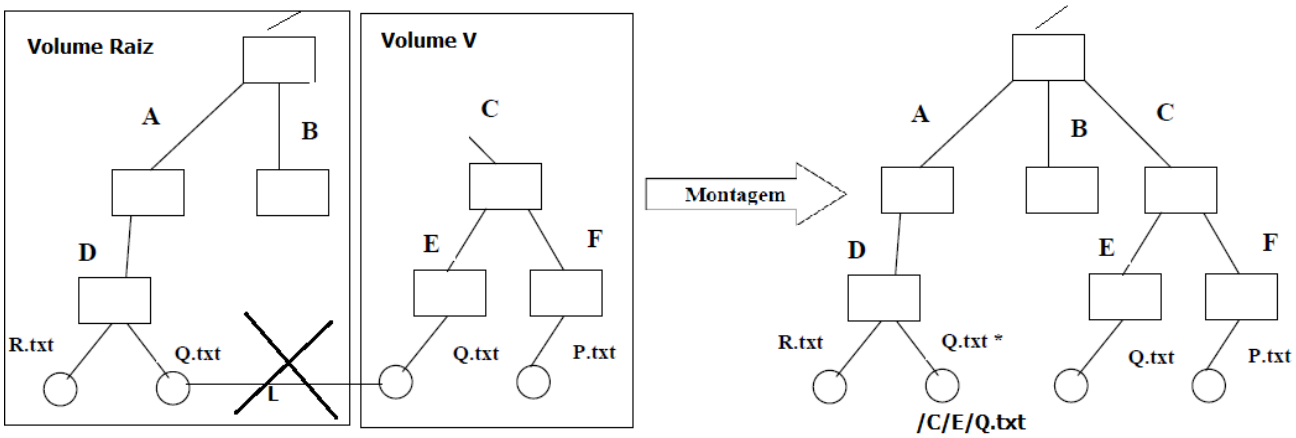
O nome completo do arquivo tem o volume que o arquivo se encontra. Por exemplo, no Windows é C:\Arquivos de Programas\Office\word.exe.

Mas, nem todo SO funciona assim é o caso do Unis e do Z/OS. Quando o usuário o usa não informa o nome do volume diz só o nome do arquivo.

LINK SIMBOLICO

Conforme mostra o lado esquerdo da Figura abaixo não é possível fazer um link entre volumes, pois o INODE é único por volume. Não tem como referenciar um arquivo a um INODE que está em outro volume. O Link Simbólico foi criado para possibilitar a criação de link entre volumes. Ele não é mapeado no INODE de origem, pois o INODE não consegue acessar o outro volume. Essa estrutura é igual ao atalho do Windows.

No UNIX não tem como se saber qual é o volume, pois não se usa o volume explicitamente. O usuário consegue usar o arquivo P.txt do Volume V através de uma operação chamada de Montagem. A Montagem faz com que o Volume V seja montado na arvore raiz (Volume Raiz) que o usuario vê atualmente. Conforme mostra o lado direito da Figura abaixo.



O símbolo * é a flag indicadora que fica no INODE que diz que o arquivo é um Link Simbólico.

Essa abordagem do Unix é bem diferente do Windows e tem uma vantagem que os arquivos/diretórios não ficam presos ao volume. E são movidos de um volume para o outro sem precisar mudar o nome completo do arquivo. No Windows para mudar um arquivo de volume é preciso mudar o nome completo do arquivo, pois dentro do nome do arquivo contem o volume que o arquivo se encontra.

O **Link Simbólico** é um arquivo que foi criado no Diretório raiz que tem como conteúdo o caminho do arquivo que se deseja acessar em outro Diretório que está em outro volume. Ele é um arquivo especial assim como o Diretório. Todo arquivo simbólico é sinalizado com a flag *. O SO trata esse arquivo de forma diferente e meio que engana o usuário.

Se apagar o arquivo original o Link Simbólico fica "preso no ar". Não é igual ao Link que continua funcionando enquanto tiver link sinalizado no INODE. Então se tentar usar o arquivo dará erro de arquivo não encontrado. O uso é igual ao link normal, mas a diferença é quando apaga.

Resumindo o Link Simbólico é um pequeno arquivo que aponta para outro arquivo no sistema de arquivos. Um link simbólico pode apontar para um arquivo em qualquer lugar, seja no próprio sistema de arquivos onde ele está localizado, seja em outro sistema de arquivos e, até mesmo em sistemas de arquivos remotos, como NFS, por exemplo. Podem também apontar para diretórios. Por ser um arquivo, um link simbólico ocupa pouco espaço no sistema de arquivos.

Ele pode ser feito entre volumes diferentes (INODE em locais diferentes) é criado um arquivo no diretório que é o caminho para chegar ao arquivo. Ao tentar abrir esse tipo de arquivo, o SO verifica se existe uma marcação especial. Então o SO, sem que o usuário saiba, abre o arquivo, lê o arquivo, mas na verdade faz a abertura de um arquivo que está em outro volume.

Observações:

- É parecido com o atalho do Windows, mas este só funciona para o Explorer, enquanto o mecanismo acima, no Unix, é controlado pelo SO, diferente do Windows que o mecanismo de atalho é controlado pelo Explorer e não pelo SO.

ATALHO X LINK

O link lembra o Atalho que existe no Windows, mas link e atalho são diferentes. Se tem o arquivo e o atalho do arquivo se o arquivo é apagado o atalho acaba. Ou seja, o atalho so existe enquanto existir o arquivo original. No caso do link isso não acontece. O atalho é feito fora do núcleo do SO e não existe o link que prende o arquivo ao INODE.

LINK SIMBOLICO X ATALHO

A diferença entre o Link Simbólico (do Unix) e o Atalho (do Windows) está em quem implementa essa estrutura. No Unix quem implementa o Link Simbólico é o núcleo do SO. Logo, sempre funcionada, pois é o núcleo que faz. No Windows não é o SO que faz, mas o Explorer (um programa importantíssimo com o qual o usuário interage para fazer as coisas no SO). Se usar com uma DLL do Windows o atalho funciona, mas se usar por fora o atalho não funciona porque um programa normal não sabe interpretar um atalho.

Digamos que se digita na linha que comando Notepad \B\Q.link. O atalho do arquivo não vai funcionar. Será aberto o Notepad com o conteúdo do arquivo local com o caminho para o arquivo do outro volume: \C\E\Q.txt. Isso acontece porque o atalho não funciona através da linha de comando somente pelo explorer ou pelas DLL do Windows, mas por fora não funciona, pois o núcleo do SO não sabe onde está. Ele sabe que existe apenas um arquivo no volume local.

O Windows antigo não tinha Link Simbólico, mas o Windows Novo tem e é usado tanto para Diretório (tem outo nome Jump) quanto para arquivo. No Unix é link simbólico para arquivo e diretório.

6) CHAMADAS AO SO PARA UTILIZAR ARQUIVOS

As chamadas básicas ao SO utilizadas por arquivos pelo UNIX são:

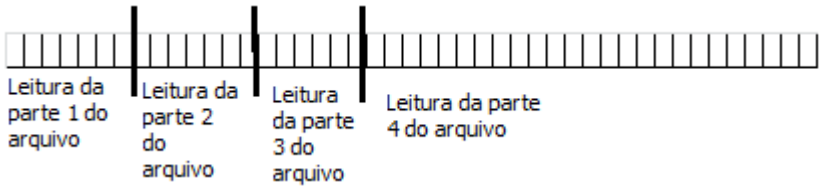
- **OPEN (NOME_ARQUIVO, MODO_ABERTURA);**
- **READ (FILEDESCRIPTOR, END_VARIAVEL, QTD_BYTES);**
- **WRITE (FILEDESCRIPTOR, END_VARIAVEL, QTD_BYTES);**
- **LSEEK (FILEDESCRIPTOR, NOVA_POSICAO, A_PARTIR_DE_ONDE);**
- **CLOSE (FILEDESCRIPTOR);**

Um arquivo que tem 10 GB não caberá na memoria, por exemplo, vídeo. Conseguimos ler o arquivo utilizando algum mecanismo para trazer este tipo de arquivo para memoria.

Nesse caso o SO tem que permitir que o programador carregue parte do arquivo para memoria e não ele todo. O mecanismo exibe o vídeo aos poucos para o usuario, pois carrega partes do vídeo na memoria quando uma parte termina de ser vista ele pega a outra parte e assim por diante até o final do vídeo.

Outra coisa importante é que um conteúdo que está no arquivo no Disco não é utilizado pela CPU. A CPU não tem como utilizar os dados que estão no arquivo. Os dados que a CPU utiliza ou estão nos Registradores ou estão na Memoria. É por isso que para ler um arquivo é preciso trazer primeiro ele para memoria.

ACESSO SEQUENCIAL



Chamada fictícia:

LER (nome_arquivo, pos_inicial,endereço_variavel_p_receber_o_conteudo, qtd_bytes_a_serem_lidos)

ler("dados.dat", 0, &var, 5.000.000); // Leitura da parte 1 do arquivo
// Uso o conteúdo de var para alguma coisa.
ler("dados.dat", 5.000.000, &var, 5.000.000); // Leitura da parte 2 do arquivo
// Uso o conteúdo de var para alguma coisa.
ler("dados.dat", 10.000.000, &var, 5.000.000); // Leitura da parte 3 do arquivo
// Uso o conteúdo de var para alguma coisa.
ler("dados.dat", 15.000.000, &var , 5.000.000); // Leitura da parte 4 do arquivo
// Uso o conteúdo de var para alguma coisa.

Essa é uma chamada fictícia, pois as chamadas reais não são assim. A chamada fictícia tem duas ineficiências (Problemas) que são: Validação do Arquivo e o Controle da Próxima posição a ser lida. A chamada é responsável por verificar se o arquivo existe ou não. Com a chamada fictícia isso não é feito. Se o arquivo não existir a chamada retorna um erro.

Da forma que a chamada fictícia é feita passando o nome do arquivo a validação para saber se o arquivo existe ou não tem que ser feita a cada chamada. No exemplo apresentado acima, o SO terá que verificar em todas às vezes se o arquivo existe ou não. Para evitar esse problema foi criado o conceito de **abertura de arquivo**.

ABERTURA DO ARQUIVO

O SO tem que testar se o arquivo existe, caso não exista ele retornará um erro. Existe uma chamada que é a abertura do arquivo que testa se o arquivo existe, ela devolve o identificador do arquivo. A abertura do Arquivo verifica se o arquivo existe e garante que o arquivo não será apagado enquanto estiver aberto.

Chamadas Reais (Unix):
ABRIR (nome_arquivo, modo_abertura)
LER (var_nome_arquivo_apos_abertura, qtd_bytes_a_serem_lidos, endereço_var_p_receber_o_conteudo)

MODO_ABERTURA → Somente Leitura, Leitura e Escrita, Somente Escrita)

A chamada de leitura real não passa como parâmetro o nome do arquivo, mas sim a variável com o numero inteiro (fd) que está com o arquivo aberto.

```
Int fd; //file descripter
//identificador de arquivos aberto
fd = open ("dados.dat",  modo_abertura)
read(fd, &var, 5.000.000)
// Uso o conteúdo de var para alguma coisa.
read(fd, &var, 5.000.000)
// Uso o conteúdo de var para alguma coisa.
read(fd, &var, 5.000.000)
// Uso o conteúdo de var para alguma coisa.
```

Nas três leituras realizadas acima o SO não precisa verificar se o arquivo existe. O SO verificará somente se o arquivo existe apenas na primeira chamada que é a open, pois esta chamada de abertura garante que o arquivo existe e continuará existindo enquanto estiver aberto.

A Chamada ao Sistema open é diferente da opção de Menu Arquivo/Abrir. A chamada OPEN apenas abre o arquivo. A opção do Menu o programa não faz apenas a chamada OPEN. Ele faz a chamada OPEN e pelo menos um READ.

Outro problema da chamada fictícia é o controle da próxima posição a ser lida, pois na chamada fictícia é reponsabilidade do programador. Isso é muito trabalhoso.

Dois processos podem abrir o mesmo arquivo se o modo de abertura for de leitura. A leitura de um arquivo pode ser por parte em sequência ou o arquivo inteiro ou saltando, sendo que este último não é comum. Na leitura por partes é o programador que controla de quantos em quantos bytes será lido. O SO controla a próxima posição da sequência a ser lida, desta forma pode ser eliminado este último parâmetro (pos-inicial-no-arquivo) de leitura. Ele controla isto utilizando um ponteiro que indica o próximo bloco a ser lido. Assim a sintaxe de leitura fica:

LER (variável_onde_ficarão_dados_lidos, nome_arquivo, tamanho_da_variável);

Usando apenas a chamada de leitura a cada chamada ler é feito à verificação se o arquivo existe. Quando tem o "open" antes da chamada de leitura "read" é verificado uma única vez, apenas no arquivo.

O sistema operacional tem que testar se o arquivo existe, caso não exista ele retornará um erro. Existe uma chamada que é a abertura do arquivo que testa se o arquivo existe, ela devolve identificador do arquivo.

Exemplo int identificadorarq;

```
identificadorarq = abre(nomearq, modo-de-abertura);
lerB(identificadorarq, variável-onde-ficarao-dados-lidos, tam-variavel);
ler(identificadorarq, &b, 100);
close(identicadorarq);
onde: modo-de-abertura pode ser: Para ler, Para escrever e Para ler e/ou escrever.
```

No Word ao abrir um arquivo não é só a chamada open que é feita. Na verdade é a chamada open seguido da chamada read se o arquivo for pequeno, se o arquivo for grande são vários reads, a cada scroll é feito um read. No wordpad a leitura é feita de uma só vez.

Existe outra chamada que permite que seja lido o arquivo saltando que é o LSEEK. Ela muda a posição corrente para a posição desejada, ou seja, altera a posição corrente do arquivo.

As chamadas das bibliotecas em C que utilizam a biblioteca padrão não são as apresentadas acima, são chamadas de mais alto nível, e tem dois propósitos:

1) Formatar o conteúdo lido e;

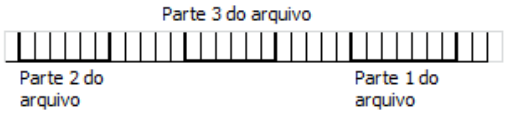
2) Otimizar o conteúdo lido, pois é feito a leitura do arquivo a mais do que a quantidade informada na linha da instrução, e armazena em um buffer interno, para evitar chamadas ao sistema operacional, já que a chamada ao sistema operacional tem um custo de tempo, pois tem que mudar o modo de operação. Ou seja, essa chamada lêem a mais.

Lembrando que o sistema operacional lê apenas seqüência de bytes. Quanto ao lseek é uma chamada de mais alto nível.

Sintaxe: lseek (identificadorarq, nova-pos, a-partir-de-onde);
a-partir-de-onde : 1) a partir do início do arquivo ou;2) a partir da posição corrente ou; 3) a partir do fim do arquivo.

Exemplo: lseek(identificadorarq, 0, dofimdoarq) => o ponteiro vai para o fim do arquivo; Este exemplo é interessante quando se quer adicionar mais informações a partir do fim do arquivo.

ACESSO RANDOMICO

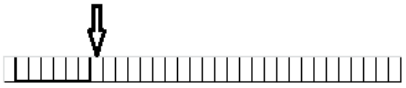


Uso mais comum é a leitura em pedaços de arquivo, ou seja, em sequencia. Outro uso é a leitura em pedaços de arquivo não sequencial (randômico). Esse tipo de leitura é típico em arquivos que guardam dados. Por exemplo, arquivos de empregados de uma empresa. Então em um pedaço desse arquivo encontra-se a matricula 378. O programa buscará e lerá no arquivo somente o registro da matricula 378.

E menos frequente esta opção e se chama Acesso Randômico (ou aleatório), mas não é bem aleatório na verdade e não sequencial. Dado que o acesso é feito por partes é mais comum. Então é necessário facilitar a vida do programador que está lendo esse arquivo para isto existe outro conceito que é o a **posicação atual do arquivo aberto**.

POSICAO ATUAL DO ARQUIVO ABERTO

O SO terá uma informação para cada arquivo aberto dizendo qual é a posição que vai inserir na próxima vez que for feita uma chamada de leitura.



Quando o arquivo é aberto à posição inicial é o byte zero. Então quando a chamada de leitura é feita. Ela lerá a partir do zero. Nesse caso a posição atual sempre apontará para o 1º byte que ainda não lido do arquivo. Com isso não será preciso se preocupar com o próximo pedaço do arquivo que precisa ser lido. O SO faz esse controle através da posição atual do arquivo aberto. Assim, o programador não precisa se preocupar com isso.

Quando utilizar Acesso Sequencial e Acesso Randômico? Depende do tipo de uso do arquivo. Por exemplo, assistir um vídeo. Pode ter um acesso sequencial ou um acesso randômico.

IMPLEMENTAÇÃO DO ACESSO RANDOMICO

LSEEK(FD, NOVA_POSICAO, A_PARTIR_DE_ONDE)

LSEEK → Muda a posição atual do arquivo. Altera a posição corrente do arquivo.
A_PARTIR_DE_ONDE → Do inicio (SEEK_BEGIN), Posição atual (-x ou +x), Do fim (SEEK_END)

```
lseek(fd, 0, SEEK_BEGIN);
read(fd, &var, 5.000.000);
close(fd);

fd=open("a.log", ...); // Criação do arquivo de log
lseek(fd, 0, SEEK_END);  // Aumenta o tamanho do arquivo.
write(fd, nova_linha_do_log, 1.000);

fd=open("dados.dat", O_RDWR);
lseek(fd, 0, SEEK_END);
write(fd, &var, 500);
```

Toda vez que deseja mudar de posição no arquivo tem que fazer o LSEEK antes.

As linguagens de programação possuem rotinas que facilitam as chamdas ao SO chamadas de Bibliotecas. Por exemplo, as linguagens tem bibliotecas para lidar com arquivo de texto (log). Em Pascal a subrotina writeln(arq, "o valor total é: ",&total); Está encapsulado no writeln as chamadas ao SO para escrever no arquivo.

CLOSE → A chamada close é feita depois que se deixa de utilizar o arquivo. Ela indica para o SO que o arquivo não será mais utilizado eo SO pode fechar. Quando um arquivo é aberto para leitura a chamada de leitura tem que ter um controle que diga que um numero é referente ao arquivo, por exemplo, o fd=5 é referente ao arquivo "dados.dat". A cada abertura de arquivo é criada uma estrutura de controle para o arquivo na memoria. O CLOSE é para avisar o SO para liberar o espaço de memoria que está sendo usado pelo arquivo. Antigamente tinha um limite para arquivos abertos, por exemplo, no máximo 20. Logo, o proposito do CLOSE é liberar a memoria interna do SO.

No caso do Windows tem outro proposito além desse que é o acesso simultâneo ao arquivo. Quando o arquivo é aberto nesse SO somente o usuario que abriu o arquivo pode usar esse arquivo. Se outro usuario tentar usar esse arquivo o SO

não deixa dá erro. Logo, o CLOSE no Windows além de liberar espaço na memoria também libera o arquivo para outra pessoa usar.

O CLOSE não garante que o conteúdo do arquivo foi salvo no disco. A chamada WRITE também não garante que o conteúdo do arquivo foi salvo no disco. Ela só indica para o SO que quer salvar, pois o salvar de fato não ocorre na chamada do WRITE. A ideia não é salvar toda hora no disco porque gasta tempo. Então é melhor deixar que o SO decida quando salvará no disco as alterações realizadas.

RESUMINDO AS CHAMADAS AO SO

OPEN → Verifica se o arquivo existe, se o usuário tem permissão de usar o arquivo e carrega em memória a estrutura de controle do arquivo.

READ → Recebe como parâmetro o verificador do arquivo aberto, que é retornado pelo open. Ler e coloca na memória o pedaço. Faz em Baixo nível e trabalha com bytes

WRITE → Parecido com o READ faz a coisa inversa. Se escrever depois do fim do arquivo aumenta o tamanho do mesmo.

CLOSE → Avisa ao SO que o arquivo aberto não será mais usado, isto é, será fechado. Então a estrutura de controle daquele arquivo será liberada. No caso de uso simultâneo de arquivo ele ajuda.

LSEEK → Altera a posição corrente para fazer leitura randômica. Como se faz a abertura de arquivo: primeiramente verifica se o arquivo existe (fd=open...).

```
Int fd;
Fd = open ("dados.dat", o_rdonly);
For (i=0; i < 10240 (10kb); i++)
{
Read (fd, &c, 1( qt de 1 byte a ser lido)) }
Close (fd);
```

Pode-se ter arquivos de vários GB e não faz muito sentido ler todo o conteúdo desse arquivo de uma vez só. É necessário ter chamadas para ler partes de um arquivo. Essas chamadas poderiam existir no SO mas seriam menos eficientes. Ex.: ler um arquivo em partes e processá-lo. Em C:

```
ler (&a, "dados.dat", 100, 0) □ lê os 100 primeiros bytes.
// Usa o dado
ler (&b, "dados.dat", 100, 100) □ lê os 100 bytes seguintes.
// Usa o dado
ler (&c, "dados.dat", 100, 200) □ lê os 100 bytes seguintes.
```

E necessário verificar se o arquivo dados.dat existe. No exemplo acima, a cada chamada o SO precisa ver se o arquivo existe. Se estivesse num loop, o SO teria que ver se o arquivo existe toda hora. É razoável que o SO verifique só uma vez se o arquivo existe ou não, a fim de evitar que o SO faça a verificação toda vez que tentar ler/escrever o arquivo.
Para verificar, o SO precisa fazer uma chamada de abertura.

```
Ex.: int fd /* fd = file descriptor */
fd = open (nome_arq, modo_abertura)
read (fd, var_onde_ficarão_os_dados lidos, tam_var)
read (fd, var_onde_ficarão_os_dados lidos, 300)
```

Não é mais necessário informar o nome do arquivo. Somente a variável que referencia o arquivo. A chamada **OPEN** impede que o arquivo seja apagado enquanto o mesmo está aberto. É necessário informar o modo de abertura do arquivo (ler, escrever ou ler/escrever).

Para fechar um arquivo, usa-se o comando close(fd). O comando close não serve somente para permitir que o arquivo possa ser alterado novamente.

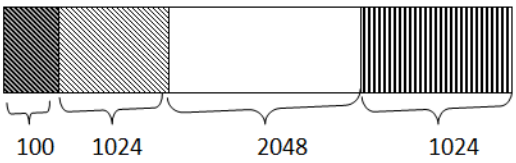
Quando o arquivo é aberto, o SO começa a guardar informações desse arquivo e, com isso, gasta-se memória. Se os arquivos nunca fossem fechados, a memória estouraria por falta de espaço livre devido ao grande volume de informações gerados para cada arquivo.

7) ARQUIVO ESPARSO

É um arquivo com espaço em branco e que ao ser salvo em disco salva apenas a parte com informação. A parte em branco não é salva. Ao ler vai ser lido com um monte de zero pelo SO. É equivalente a matriz esparsa. Não pode ter arquivos esparsos em alocação encadeada, porque tem que ter encadeamento dos blocos em que faz parte o arquivo.

O sistema operacional completa o espaço em branco, este pedaço tem que existir pela definição de arquivo da visão lógica, que um arquivo é uma sequência de byte. Mas, não existem fisicamente estes espaços em branco no disco, ou seja, o arquivo tem 5 Kbyte, mas fisicamente no disco ele ocupa 3 Kbyte. A vantagem disso é não ocupar espaços desnecessários no disco com espaço em branco.

Lembrando que a visão lógica de um arquivo é um "array" de byte. Segue abaixo o esquema do arquivo como um array de byte, sendo a primeira posição o tamanho inicial do arquivo 100 bytes, segunda posição houve crescimento do arquivo para mais 1024 bytes, a terceira posição está em branco, pois houve um deslocamento de 2048 bytes para depois do fim do arquivo, ficando 2048 bytes de espaço em branco e a quarta e última posição são 1024 bytes de informação.



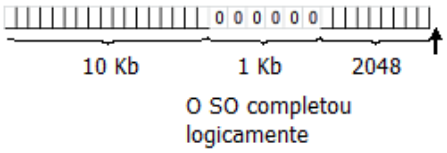
Como no Arquivo esparso o espaço é completado com 0 não existe no disco, existe logicamente. Se tentar ler esse pedaço não dará erro, retorna 0. Isto é feito para manter o conceito de que o arquivo é uma sequência de bytes.
Fd=open ("data.dat", o_wronly);
Lseek (fd,1024,seek_end)=(fd, nova_posicao, a_partir_de_onde);
Write (fd,&var,100);

A partir do início do arq ou posição atual ou fim do arquivo (append – seek_end). No arquivo esparso tam_logico > tam_fisico. Arquivo esparso e matriz de 1024 X 1024 (1MB).

Implementação: ao invés de fazer um write na linha, faz-se Lseek (fd, 1024 x4, seek_cur). Assim pula-se a linha que só contém zero. 4 pq cada elemento tem 4kb (4kb X 1MB = 4M).

Conforme mostra a Figura abaixo, o tamanho inicial (tamanho logico) do arquivo é 13 Kb, mas na realidade o espaço que ele está gastando no disco é menor, 12 Kb. Esse arquivo é diferente ele tem um "buraco" no meio. Se o usuario fizer uma leitura randômica para ler o "buraco" retornará para ele um monte de zeros. Esse tipo de arquivo é chamado de Arquivo Esparso.

fd=open("dados.dat", O_RDWR);
lseek(fd, 1024, SEEK_END);
write(fd, &var, 2048);

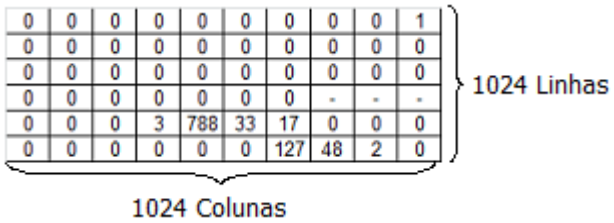


Arquivo Esparso acontece quando o Tamanho Logico > Tamanho Físico (ou espaço gasto no disco). Isso não é normal. O normal é o Tamanho Fisico ser maior que o Tamanho Logico (Arquivo Normal = Tamanho Logico <= Tamanho Fisico).

Existe uma unidade de salvamento no disco. Digamos que essa unidade seja de 1 Kb. Tenho um arquivo que tem 1Kb + 1Kb + 1Kb (mais não está usando todo espaço desse 1 Kb). Mas, como a unidade de salvamento é de 1 Kb. Será gasto com esse arquivo 3 unidade (Blocos). No final do ultimo bloco tem um Fragmento Interno.

Nesse arquivo temos que o Tamanho Logico = 2500 e o Tamanho Físico = 3 Kb. A divisao em blocos e o vazio que existe o usuario não vê. Arquivo é uma coisa que só interessa para o SO.

O Arquivo Esparso não serve para muita coisa. Digamos que temos uma matriz(1024x1024) o elemento dela é um inteiro. Pode ser que a matriz tenha na maioria de suas celulas iguais a zero e poucas celulas com valores diferentes de zero. Pode-se utilizar a capacidade de não salvar do Arquivo Esparso. Para evitar gastar espaço no disco salvando as linhas que não contem conteudo diferente de zero.



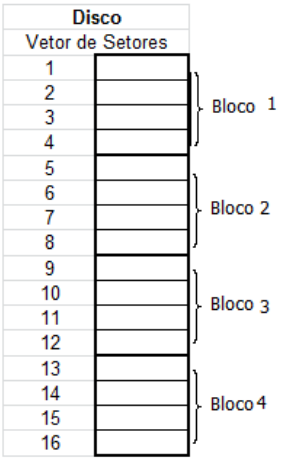
Exemplo de codigo para salvar o conteudo na matriz:

```
For (i = 0; i< 1024; i ++){
     If(linha_so_tem_zeros(linha[i])
        lseek(fd, 4096, SEEK_END);
     else
       write(fd, linha[i], 4096);
close(fd);
```

No Unix o Tamanho Logico do arquivo é armazenado no INODE. No Windows é em uma tabela equivalente ao INODE.

8) ALOCAÇÃO DE ARQUIVO NO DISCO

Problema: Onde o arquivo está no disco? Não é de quem está programando nem usando. Isto é um problema só do SO. Qual a divisao que o SO faz do Disco? O HW entrega para o SO o disco como um Vetor de Setores.



O problema é que o setor tem um tamanho pequeno porque quando foi inventado o valor determinado foi de 512 bytes. Não se aumentou atualmente porque tem medo que o impacto seja muito grandes nos SOs.

Para fazer a Alocao De Arquivos No Disco agrupa-se os setores em Blocos. Por exemplo, um bloco é formado por 4 setores. O Volume (Disco) é visto como Vetor de Blocos. O normal hoje em dia, é um bloco ter o tamanho de 4 Kb, ou seja, possui 8 setores.

Alocação de arquivo é para saber onde está a localização do arquivo no volume do disco. Existe uma visão logica do disco, que é composto por unidades, e cada uma tem um número especifico. No caso de volume a unidade não é mais chamada de setor, é chamada de bloco.

E cada bloco pode ter agrupado um ou mais setor, no mínimo um. Então, um volume é visto dessa forma, um conjunto de setores contínuos. Tem uma parte que guarda os controles e os arquivos. A questão é como os arquivos são postos dentro do disco em diferentes volumes.

A Alocação de Arquivos no Disco pode ser:

- Contínua
- Encadeada
- Encadeada com FAT
- Indexada
- Por Extensão

ALOCAÇÃO DE ARQUIVOS CONTÍNUA

A forma mais simples de organizar arquivos dentro do volume, que é a mais antiga, é chamada de **Alocação Contínua**. A locação continua tem uma regra forte, que obriga os arquivos estarem contínuos dentro do volume. Estarem em um numero SEQUENCIAL de blocos. O SO também guarda os dados de controle do arquivo, nome, bloco inicial e a quantidade de blocos.

Pela regra da continuidade existe uma limitação grande, se não tiver como expandir para mais blocos, e o próximo bloco já estiver ocupado, não vai poder salvar um arquivo maior, pois não consegue aumentar o tamanho do arquivo, só se tiver espaço no bloco seguinte. Outra limitação é a Fragmentação.

Para cada arquivo existe um registro de controle de alocação: Nome, Bloco Inicial, Quantidade De blocos. Uso de um conjunto de blocos seguidos no disco.

# bloco	Arquivo
1	A
2	A
3	A
4	A
5	A
6	B
7	B
8	C
9	C
10	C
11	C

Tabela de controle		
Nome	Bloco inicial	Número de blocos
A	1	5
B	6	2
C	8	4

Vantagens da Alocação de Arquivos Contínua

- Simples de gerenciar;
- Mais rápida, pois os blocos seguintes, depois de primeiro, não gastam tempo de seek e latência;

Desvantagens da Alocação de Arquivos Contínua

- Sujeita a Fragmentação Interna;
- Os arquivos não podem crescer;

ALOCAÇÃO DE ARQUIVOS ENCADEADA

Quando se criava um arquivo, antigamente, se criava um arquivo o quão grande poderia ser, para assim não ter problemas futuros, mas ocupava espaço que não estava sendo usando ainda. Atualmente, se pode alocar arquivos em diferentes (blocos distantes) do volume isso é chamado de **Alocação Encadeada**.

É como se fosse uma lista encadeada. É guardada no diretório, o nome do arquivo e o bloco inicial. E os demais blocos do arquivo ficam armazenados em uma lista encadeada, e esse ponteiro que aponta para próximo bloco, fica no final do arquivo, que aponta para o próximo bloco, visível apenas para o SO. Quando chega o final do arquivo põe um numero negativo (-1) que indica que não tem próximo.

Um arquivo dessa forma com pedaços separados está fragmentado. Este arquivo funciona, o usuário não vê que está Fragmentado, as gera perda de desempenho, vai gastar um tempo maior para ler esse arquivo todo.

Enquanto tiver sequencial o disco vai ler direto, mas vai ter que pular e vai perder um tempo para ler, ele gasta um ciclo para começa a ler, como estão em sequencia, é só transferência de conteúdo, mas quando for para um bloco não sequencial, vai gastar um ciclo novo, uma latência nova, esperar passar debaixo da cabeça, no caso mais grave um ciclo.

A Desfragmentação do disco torna os arquivos contínuos. A leitura do arquivo só gasta um ciclo ou latência. Num pendrive isso não faz diferença, porque não tem latência, não tem rotação.

A alocação encadeada é muito ineficiente no acesso randômico. Digamos que o primeiro registro que se queria consultar esteja dentro do quinto bloco do arquivo. Então começa a ler pelo primeiro bloco, não tem como ir direto para o quinto bloco, não se tem o endereço de lá. Tem que percorrer a lista encadeada, percorrer 4 vezes, não se sabe quem é o 5 bloco.

A ideia de ter o encadeamento não é guardar no bloco, é guardar num lugar separado, que conterá somente o encadeamento daquele arquivo. Então se tem um array de numero, em que se guarda nesse array, o próximo bloco de um arquivo, por exemplo, no caso do A é bloco 1, o próximo é o 2, e o próximo é o 3, e o próximo 4. E assim vai para cada um dos arquivos.

Exemplo como funciona a Alocação Encadeada

Arquivo A começa no bloco 1 o próximo bloco é o bloco 2 e assim por diante até chegar no final do arquivo. O bloco do arquivo B é o bloco 4 o próximo bloco é o 5. Cada arquivo tem um campo de controle do seu próximo bloco. O bloco 5 tem um campo de controle de próximo bloco com -1. O -1 indica que o arquivo não tem mais blocos.

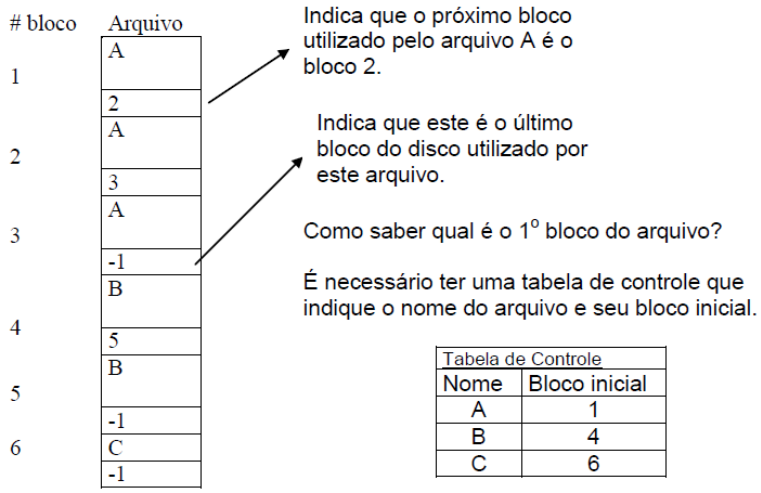


Tabela de Controle	
Nome	Bloco inicial
A	1
B	4
C	6

Na alocação encadeada os arquivos podem estar contínuos, mas não precisam estar, no caso do B o bloco está contínuo.

É interessante realocar arquivos em blocos vizinhos, pois o SO vai ser rápido. Tentam fazer alocação contínua se for possível. Os campos de controle obrigam que o arquivo seja contínuo, o arquivo não pode ser descontínuo.

Se cada bloco do arquivo tiver 1K então o arquivo que vai do byte 1 até o byte 5120. A visão do usuário é que o arquivo é uma sequência de bytes. Não o importa se o arquivo é contínuo ou não o que importa é o tamanho do arquivo. Quanto à velocidade, se o acesso for randômico é interessante que o arquivo esteja contínuo.

Um arquivo estar contínuo é mais fácil o que importa é que os blocos não sejam saltados. Dado um arquivo no bloco, o próximo bloco esteja no mesmo cilindro. Se isso for verdade, a leitura do próximo bloco vai ser rápida, o ideal, é que seja no próximo setor então vai ser mais rápido ainda, não precisa nem rodar o disco todo, mas se está no mesmo cilindro é uma grande coisa.

O programador pode ver o disco assim; A visão lógica contínua; 5 blocos; O usuário vê 5120 páginas. A alocação encadeada não é boa para acesso randômico, ou seja, que leia várias partes do arquivo, porque isso tem que andar em diferentes blocos desse arquivo, tem que ficar fazendo saltar os blocos para ler parte do arquivo para fazer leitura.

Neste caso da alocação encadeada é ruim porque para saber onde está certo dado do disco tenho que saber onde está o anterior, ou seja, precisa ler todos os blocos que vem antes. A alocação encadeada não existe na prática.

Portanto, na alocação encadeada no final de cada bloco do disco, existe um campo de controle que diz o número do próximo bloco do arquivo.

Qualquer arquivo pode crescer no disco, desde que haja blocos físicos (blocos do disco) disponíveis. O problema dessa solução é que, se existir um arquivo de 10.000 blocos e o usuário quiser ler uma informação que esteja no bloco 9.999, deverá percorrer toda a lista até chegar ao bloco desejado, gastando-se muito tempo.

Os arquivos podem crescer sem problema, desde que haja bloco disponível. Na alocação contínua, a leitura é randômica e logo, é muito mais rápida.

É bom que os blocos usados por um arquivo estejam próximos uns aos outros para que possam ser lidos mais rapidamente, pois se na visão lógica do disco (visto como um array) estiverem próximos, eles também estão próximos fisicamente (no mesmo cilindro ou nos cilindros próximos).

Vantagens da Alocação de Arquivos Encadeada

- Conseguimos crescer os arquivos;

Desvantagens da Alocação de Arquivos Encadeada

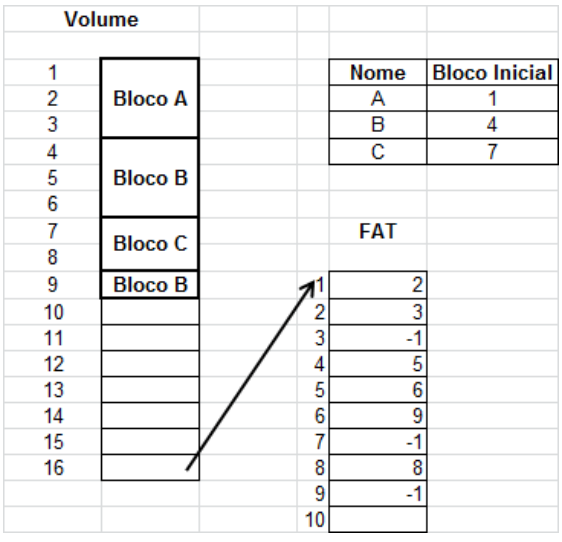
- É ineficiente no acesso randômico, pois para chegar a um determinado bloco no centro do arquivo precisamos ler todos os blocos anteriores a ele.

ALOCAÇÃO DE ARQUIVOS ENCADEADA COM ESTRUTURA DE CONTROLE SEPARADA (FAT)

Alocação encadeada com estrutura de encadeamento separada: não mantém o encadeamento no fim de cada bloco e sim reserva um bloco para controlar o encadeamento (nome, bloco inicial) FAT – FILE ALLOCATION TABLE. Está em lugar fixo no disco e contínuo.

O problema é que a FAT fica grande e acaba ocupando muita memória. E como ela é muito grande o sistema operacional tem ficar salvando parte da FAT em disco, então parte fica na memória e parte fica em disco. A FAT tem a pretensão de ter em uma única estrutura de controle as informações de todos os arquivos. Não tem relação com open e close porque está na memória.

Os dados de controle (que informam qual o próximo bloco de um determinado arquivo) são colocados em uma área separada. Estes dados ficam na memória, na área reservada para o SO.



A FAT é guardada em um bloco do Volume. Usando a FAT, o único dado guardado é o numero do próximo bloco usado por determinado arquivo. Com isso, só é necessário percorrer o array de encadeamento (FAT) que está na memoria sem precisar ler os blocos do disco para chegar ao bloco desejado.

O encadeamento deve ser percorrido, mas não envolve a leitura de todos os blocos no registro, e sim uma leitura de um array na memória. Quando chegar ao elemento do array que contém o valor do bloco do disco a ser acessado, o bloco indicado será acessado.

O tempo para ler um array na memória é muito menor do que se percorressem os blocos do disco em busca do bloco desejado. Primeiro o array é criado em disco e depois é copiado para a memória para agilizar o processo de localização de um bloco. A FAT deve ser pequena para caber na memória.

Problema: se o disco for muito grande (por exemplo, 20 GB) e tivermos blocos de tamanho pequeno (1 KB), a FAT ficaria muito grande. Se, por outro lado, quisermos que a FAT seja pequena, o bloco será muito grande e, com isso, poderá causar desperdício de espaço dentro de um bloco.

FAT 16 (16 bits por entrada na FAT)

Se usarmos um disco de 16 GB:

O Número Máximo de Entradas na FAT será de $2^{15} - 1$ (32767).

Cada bloco deve ter:

Tamanho do disco: 16 GB = 2^{10} (1 KB) * 2^{10} (1 KB) * 2^{10} (1 KB) * 2^4 (16) = 2^{34}

Número de entradas: 32767 = 2^{10} (1 KB) * 2^5 (32) = 2^{15}

Tamanho do bloco = Tamanho do Disco / Número de Entradas = 2^{34} / 2^{15} = 2^{19} (512 KB)

Se um arquivo tiver apenas 1 KB, ele ocupará 512 KB que é a menor unidade do disco (1 bloco). Logo, se usarmos um bloco muito grande, a probabilidade de haver desperdício de espaço em disco (Fragmentação Interna) será muito grande.

Já o tamanho da FAT seria de 32767 entradas * 2 bytes (16 bits que é o tamanho da entrada da FAT) = 64 KB

FAT 32 (32 bits por entrada na FAT)

Para resolver o problema de desperdício de espaço por bloco, a Microsoft criou, posteriormente, a FAT 32. Cada entrada na FAT 32 tem 32 bits.

Descontando o bit de sinal, existem 2^{31} entradas na FAT = 2 G entradas.

Tamanho do disco: 16 GB = 2^{10} (1 KB) * 2^{10} (1 KB) * 2^{10} (1 KB) * 2^4 (16) = 2^{34}

Número de entradas: 2 G = 2^{10} (1024) * 2^{10} (1024) * 2^{10} (1024) * 2 = 2^{31}

Se quisermos um bloco de tamanho 1KB, o número de blocos no disco será de:

Número de blocos no disco = Tamanho do disco / tamanho do bloco = 16 GB /1 KB = 16 M blocos

Nesse exemplo, o problema é o tamanho da FAT:

Tamanho da FAT = número de entradas da FAT * tamanho da entrada = 16 M entradas * 4 bytes = 64 MB

64 MB é um espaço grande a ser ocupado na memória.

Logo, o tamanho do bloco deve ser maior. As próximas soluções têm uma estrutura de controle para cada arquivo.

Windows FAT 32

1) Volume: 1 TB e Tamanho do Bloco: 1 KB

Qtd de blocos do volume = Tamanho Volume/Tamanho Bloco = 1 TB/1 KB= 1 G blocos

Qtd de blocos do Volume = Qtd Entradas (Registros) na FAT = 1 G

Tamanho da FAT = Qtd de Entradas na FAT * Tamanho da Entrada = 1 G * 4 B = 4 GB

2) Volume: 1 TB e Tamanho do Bloco: 128 KB

Qtd de blocos do volume = Tamanho Volume/Tamanho Bloco = 1 TB/128 KB = 2^{23} = 8 M blocos

Tamanho da FAT = Qtd de Entradas na FAT * Tamanho da Entrada = 8 M * 4 B = 32 GB

Tamanho da Entrada (Padrão Microsoft):
FAT 12 = Entrada na FAT tem 12 bits (= 1,5 Bytes). Ex.: Disquete
FAT 16 = Entrada na FAT tem 16 bits (= 2 Bytes).
FAT 32 = Entrada na FAT tem 32 bits (= 4 Bytes). Ex.: Pendrive

Essa solução não é muito utilizado, pois a FAT passou a ser muito grande e não consegue ser colocada toda na memoria por ser muito grande.

Vantagens da Alocação de Arquivos Encadeada com FAT

- É menor e fica na memória que é muito mais rápida;
- Não precisa ler todos os blocos de um arquivo no disco e sim o encadeamento na memória;

Desvantagens da Alocação de Arquivos Encadeada com FAT

- Não é boa para volumes grandes, pois tem que controlar todos os arquivos que estão no disco ficando muito grande;
- Carrega para memoria um monte de dados de controle de arquivos que não está sendo usado naquele momento.

ALOCAÇÃO DE ARQUIVOS INDEXADA (SOLUÇÃO DO UNIX)

Tem uma estrutura de controle para cada arquivo, então somente quando o arquivo A for aberto é carregado à estrutura de controle do arquivo A para memoria, diferente da FAT que carrega toda estrutura de controle que é única para todos os arquivos.

A ideia é quebrar estrutura de controle de arquivo muito grande. No Unix esta estrutura é chamada de INODE, onde ficam os dez primeiros blocos que compõe o arquivo. Ao abrir o arquivo somente o INODE é carregado para a memória. Os blocos de controle vão para a memória quando o arquivo está sendo usado.

O INODE guarda as informações relevantes do arquivo, data da criação, quem criou, e também o array que guarda os números do bloco do arquivo, no caso de um arquivo pequeno esses números cabem nesse array (10 posições = 10 blocos), no exemplo apresentado na Figura abaixo só o INODE resolve, pois o arquivo B possui menos de 10 blocos.

Arq. A	Arq. B	Arq. C
1	4	7
2	5	8
3	6	
	9	

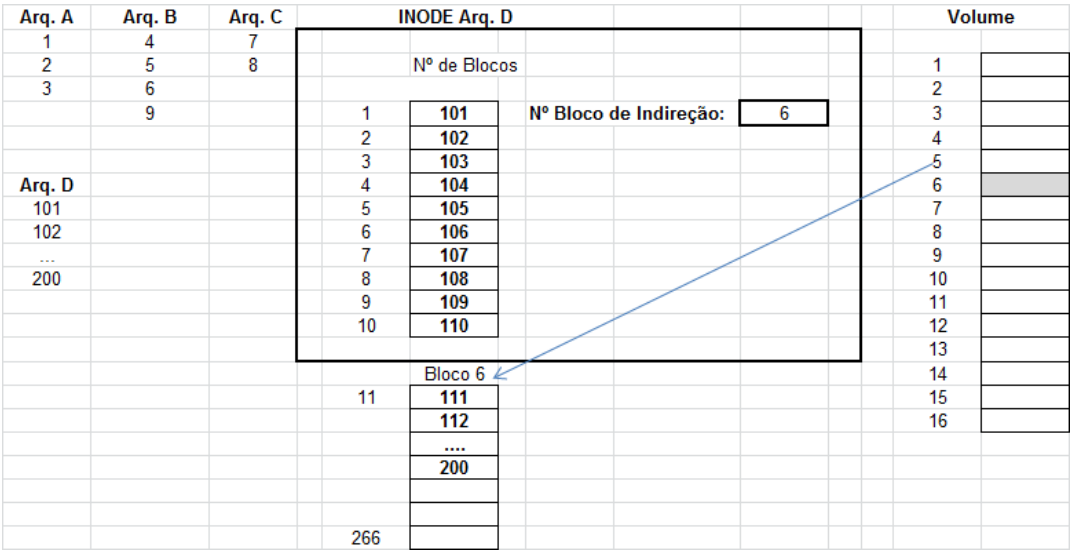
INODE Arq. B

	Nº de Blocos
1	4
2	5
3	6
4	9
5	
6	
7	
8	
9	
10	

Quando o arquivo possui mais de 10 blocos pega se um bloco vazio, para usar como continuação da lista. Esse bloco não é para armazenar dados e sim controle, este é uma continuação da lista. Como o bloco é muito grande cabem um nº grande de blocos dentro de 1 bloco.

Por exemplo, o Arquivo D possui 100 blocos. Não cabe tudo no INODE , somente os 10 primeiros blocos. Então o restante (90 blocos) é guardado em um bloco chamado de **Bloco de Indireção (Simples)**. Esse bloco aponta para um bloco do disco que esta livre é uma extensão do array que está no INODE.

Conforme mostra a Figura abaixo, no INODE do arquivo existe uma indicação que diz onde está à continuação do array: **Nº Bloco de Indireção.**

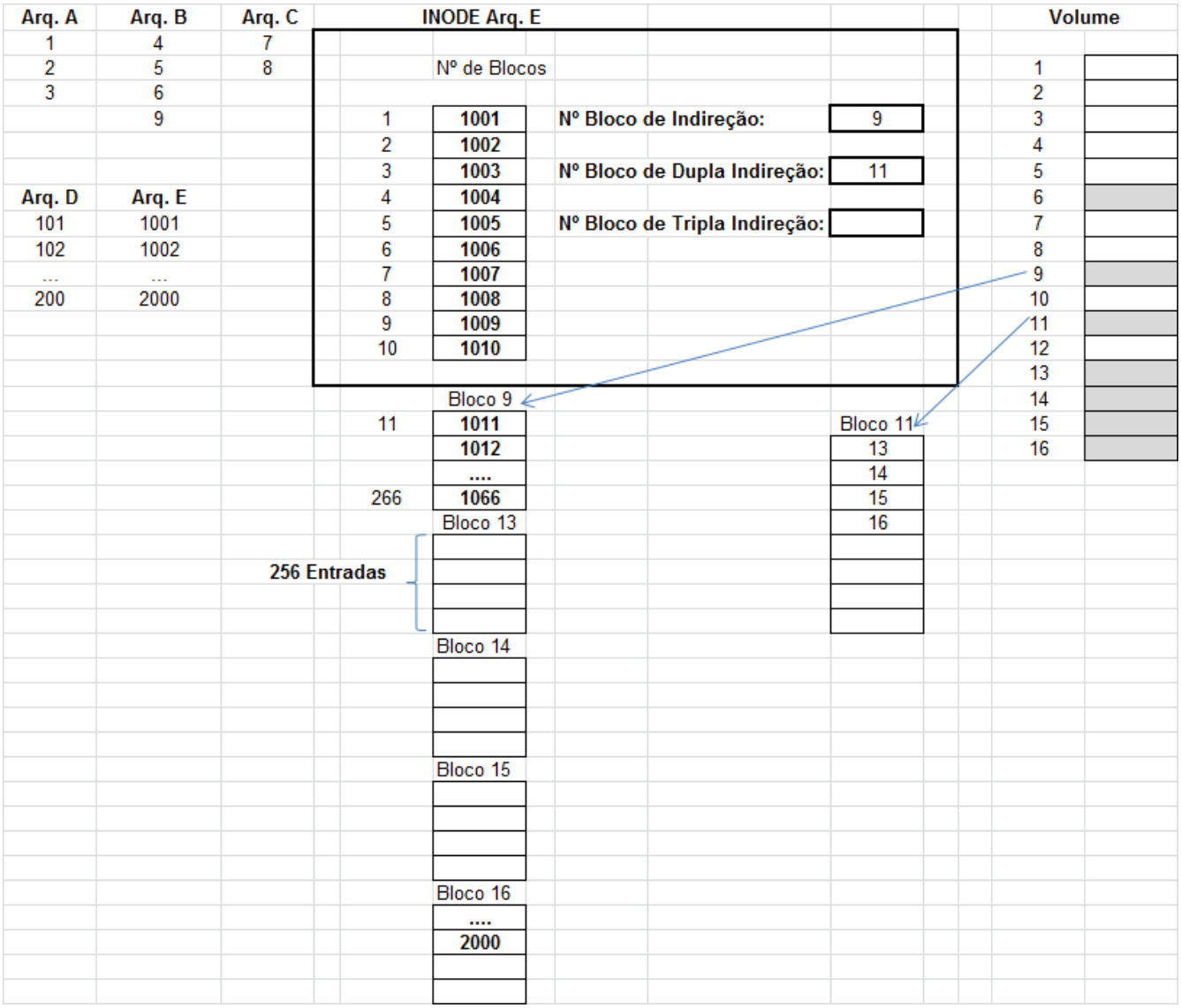


Digamos que agora temos mais um arquivo E que precisa de 1000 blocos. Nesse caso, primeiro vamos verificar se caberá no Bloco de Indireção Simples. O cálculo é o seguinte:

Qtd. de Entradas em um Bloco de Indireção = Tamanho do Bloco/Tamanho da Entrada.
Tamanho de Bloco = 1k e o Tamanho da Entrada = 4 bytes
Qtd de Entradas em um Bloco de Indireção = 1Kb/4b = 256 (Tamanho máximo de um Bloco de Indireção)

Quantidade máxima de blocos de um Arquivo = 10 + 256 = 266 blocos
Tamanho Máximo de Arquivo = Nº máximo de Blocos * Tamanho do Bloco = 266 * 1 KB = 266 KB

Logo, o Arquivo E possui mais de 266 blocos e não consegue ser guardado com o INODE + Bloco de Indireção. A solução para esse problema é o **Bloco de Dupla Indireção**. Ele armazena os **Números dos Blocos de Indireção (Simples)**.



Quantos blocos cabem no **Bloco de Dupla de Indireção**?

Qtd. de Entradas em um Bloco de Indireção = Tamanho do Bloco/Tamanho da Entrada.
Tamanho de Bloco = 1k e o Tamanho da Entrada = 4 bytes
Qtd de Entradas em um Bloco de Indireção = 1Kb/4b = 256 (Tamanho máximo de um Bloco de Indireção)
Quantidade máxima de blocos de um Arquivo = 10 + 256 + 256^2 = 65802 =~ 64 K Blocos
Tamanho Máximo de Arquivo = Nº máximo de Blocos * Tamanho do Bloco = 64 K * 1 KB = 64 KB

Para arquivos maiores que o Arquivo E do exemplo acima, ou seja, com mais de 1000 blocos existe o **Bloco de Tripla Indireção**. Ele aponta para **256 Blocos de Dupla Indireção**. E cada Bloco de Dupla Indireção aponta para um Bloco de Indireção Simples.

Quantos blocos cabem no **Bloco de Tripla de Indireção**?

Qtd. de Entradas em um Bloco de Indireção = Tamanho do Bloco/Tamanho da Entrada.
Tamanho de Bloco = 1k e o Tamanho da Entrada = 4 bytes
Qtd de Entradas em um Bloco de Indireção = 1Kb/4b = 256 (Tamanho máximo de um Bloco de Indireção)
Quantidade máxima de blocos de um Arquivo = 10 + 256 + 256^2 + 256^3 = 16.843.018=~ 16 Milhões Blocos
Tamanho Máximo de Arquivo = Nº máximo de Blocos * Tamanho do Bloco = 16 M * 1 KB = 16 GB

Conclusão

Qtd. de Entradas em 1 bloco de Indireção = Tamanho do Bloco/Tamanho de Entrada = 1 KB/4 B=256 Entradas

Qtd. máxima de blocos = 10 + 256 = 266 Blocos (Indireção Simples)

Qtd. máxima de blocos = 10 + (256*256) = 65.000 blocos (Dupla Indireção)

Qtd. máxima de blocos = 10 + (256*256*256) = 16.000.000 blocos (Tripla Indireção)

*Se quiser aumentar a capacidade de armazenamento é só aumentar o Tamanho do Bloco.

Vantagens da Alocação de Arquivos Indexada

- Cada arquivo tem sua própria estrutura de controle. Com isso, só os arquivos que estão sendo usados (abertos) é que têm suas estruturas de controle carregadas na memória.

Desvantagens da Alocação de Arquivos Indexada

-

ALOCAÇÃO DE ARQUIVOS POR EXTENSÃO (NTFS) (SOLUÇÃO WINDOWS E MAINFRAME)

A ideia da Alocação por Extensão é ter um registro de controle para cada pedaço continuo do arquivo. Um arquivo pode ter uma continuação em outra parte do disco. Foi criada pela IBM.

O Windows e o Mainframe utilizam esse tipo de alocação. Se o arquivo tiver parte dos blocos contínuos a leitura é mais rápida. A alocação por extensão é melhor do que a alocação indexada quando a maquina possui o mecanismo periódico de desfragmentação que torna os arquivos contínuos no disco.

Para ter um bom desempenho os arquivos devem estar contínuos na memoria, caso contrário será muito lento e a Alocação Indexada é melhor para arquivos não contínuos. No caso de ter pedaços contínuos a indexada é pior que o NTFS.

A Alocação por Extensão estende o conceito de alocação contínua, permitindo que o arquivo possa ser descontínuo. A ideia e se o arquivo tiver dois pedacos continuos se coloca dois registros no array de controle do arquivo. Logo utiliza-se uma extensão da ideia da Alocação Continua. Dessa forma a estrutura de controle do arquivo fica exulta, pois gasta se menos espaço para controlar.

Arquivo B

Bloco Inicial	Qtd Blocos	Nº Bloco do Arquivo
4	3	1º
9	2	4º

O Nº Bloco no Arquivo guarda o inicio do pedaço continuo do arquivo. Este campo ajuda o SO encontrar onde está o enessimo bloco do arquivo. Para encontrar, por exemplo, onde está o 4º Bloco é só somar a Qtd. Blocos + Nº Bloco no Arquivo = 4. Entao se descobre que o 4º Bloco do Arquivo B começa no Bloco 9.

A Figura abaixo mostra um exemplo de um arquivo que não é continuo. Como esse arquivo é bastante descontinuo a Alocação Indexada é melhor do que a Alocação por Extensão, pois o espaço para guardar o controle da Indexada é menor do que a por Extensão.

Arquivo X - Alocação Indexada

2
4
5
8
10
12
13
16
18
9 Números

Arquivo X - Alocação por Extensão

Bloco Inicial	Qtd Blocos	Nº Bloco do Arquivo
2	1	1º
4	2	2º
8	1	4º
10	1	5º
12	2	6º
16	1	8º
18	1	9º
7 Registros x 3 Campos = 21 Números		

Então é uma questão de sorte. Se o arquivo é continuo ou não para saber qual a estrutura é melhor. Ou seja, qual Alocação de Arquivo é melhor.

Vantagens da Alocação de Arquivos Por Extensão

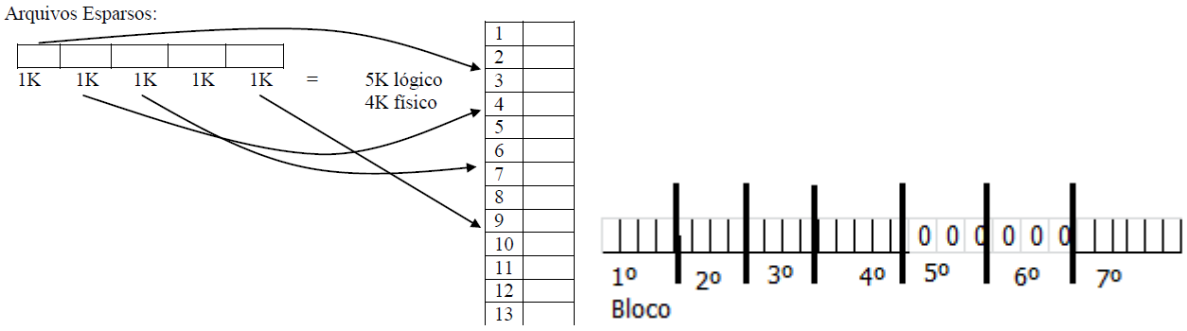
- É melhor para arquivo continuo, pois gasta menos espaço com a estrutura de controle.

Desvantagens da Alocação de Arquivos Por Extensão

- É ruim para arquivo descontinuo, pois se gasta mais espaço com a estrutura de controle.

9) <u>COMO CONTROLAR OS ARQUIVOS ESPARSOS NESTES DIFERENTES TIPOS DE ALOCAÇÃO</u>

No disco, parece que o arquivo é contínuo pois está contínuo fisicamente mas não é logicamente contínuo, é esparso. No disco os blocos podem estar contínuos, mas podem fazer parte de um **Arquivo Esparso**. Como este arquivo é controlado?



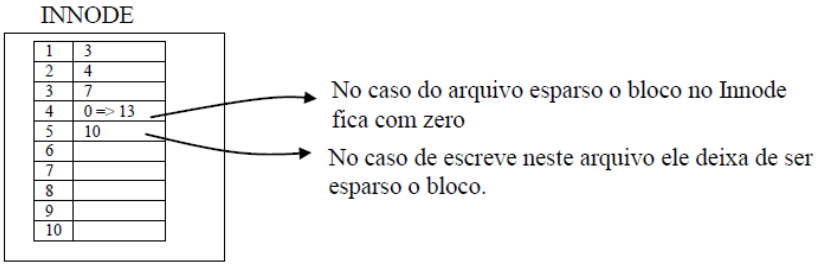

```
fd=open("dados.dat", O_RDWR);
lseek(fd, 2048, SEEK_END);
write(fd, &var, 100);
```

Na **Alocação Encadeada** não pode ter Arquivos Esparsos porque tem que ter encadeamento dos blocos em que faz parte o arquivo.

Na **Alocação Encadeada com FAT** os blocos vazios do Arquivo Esparso são sinalizados com o zero (0). Isso indica que os blocos 5 e 6 estao livres, pois não pertencem a nenhum arquivo. A FAT também é usada para controle de blocos livres. A Alocação Indexada e Por Extensão não controlam os blocos livres, pois são individuais por arquivo.

No Unix na **Alocação Indexada**, o INODE representa um bloco que não existe fisicamente marcando na entrada o valor 0 (zero). Por exemplo, logicamente um arquivo ocupa 5120 bytes, mas fisicamente ocupa somente 4096 bytes. Conforme mostra a Figura a seguir:

INODE do	Arquivo X - Alocação Indexada
1º	3
2º	4
3º	7
4º	8
5º	0
6º	0
7º	12



Como saber se o arquivo é um Arquivo Esparso? Some a **Qtd Blocos** com o **Nº Bloco do Arquivo**. Se for diferente do Nº Bloco do Arquivo da linha seguinte é porque há blocos lógicos vazios dentro do arquivo. Logo, o arquivo é um Arquivo Esparso. Conforme mostra a Figura abaixo. Se for igual ao Nº Bloco do Arquivo da linha seguinte é porque **não** há blocos lógicos vazios dentro do arquivo. Logo, o arquivo não é um Arquivo Esparso.

Arquivo X - Alocação por Extensão		
Bloco Inicial	**Qtd Blocos**	**Nº Bloco do Arquivo**
5	2	1º
7	2	4º

Bloco Inicial da Extensão	**Nº de Blocos da Extensão**	**Nº Bloco no arquivo**
3	2	1
7	1	3
10	1	4

O bloco 4 é esparso

Onde está o bloco que contém a parte esparsa? O NTFS tem na sua estrutura de controle mais um campo que indica o bloco esparso. E para descobrir a parte esparsa: soma-se o número do bloco da extensão mais o nº do bloco no arquivo e o resultado deve ser igual ao próximo valor da coluna nº bloco do arquivo. Caso contrário então o arquivo é esparso.

Qual a 2ª função do **Nº Bloco do Arquivo**? Dá mais eficiência ao NT. Acha-se com mais rapidez onde está um determinado bloco de um arquivo.

Bloco Inicial da Extensão	Nº de Blocos da Extensão	Nº Bloco no arquivo
3	2	1
7	1	3
13	1	4
10	1	5

Quando o arquivo deixa de ser esparso a estrutura de controle fica, por exemplo, conforme mostra a Figura ao lado.

Neste exemplo, o arquivo esparso no Unix, alocação indexada é melhor do que o NTFS, pois para saber onde está o bloco com a parte esparsa é só varrer o INNODE e pegar, pelo exemplo anterior, a quarta posição. Já no NTFS tem que fazer pesquisa binária.

10) GERÊNCIA DE BLOCOS LIVRES

Problema: Como o SO sabe quais os blocos que estão livres?

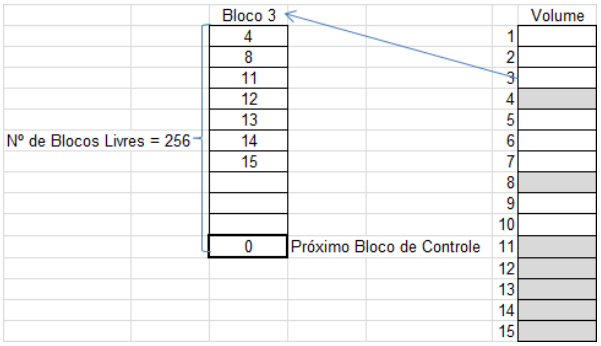
Soluções: 1) Lista de Blocos Livres
2) Vetor de Bits (Bitmap) de Blocos Livres

LISTA DE BLOCOS LIVRES

O SO verifica em todos os arquivos do disco e vê quais blocos não estão sendo utilizados por nenhum arquivo. O sistema operacional tem um campo que diz quem é o primeiro da lista, então para descobrir um bloco livre tem que percorrer toda a lista para saber se certo bloco está livre ou não.

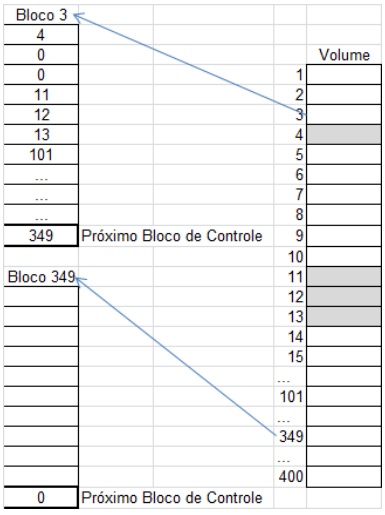
Por exemplo, o Bloco 3 é um Bloco de Controle que guarda a lista de Blocos Livres. Assim como na Alocação Indexada o Bloco tem um limite para guardar um Nº de Blocos Livres.

Qtd. Entradas em 1 Bloco = Tamanho do Bloco /Tamanho da Entrada = 1KB/4B = 1KB = 256 Entradas



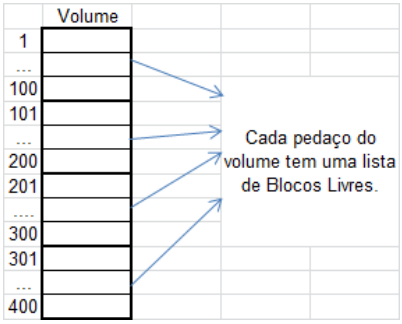
A quantidade de blocos livres que existe em um volume é muito maior do a capacidade que um Bloco de Controle tem para guardar que é igual 256 Blocos Livres. Por exemplo, existem mais de 256 Blocos Livres no volume, logo será preciso de mais de um Bloco de Controle para guardar esses Blocos Livres.

O próximo Bloco de Controle é guardado na última posição do Bloco de Controle, então temos 255 entradas para guardar os Blocos Livres e 1 entrada para guardar o próximo bloco de controle. Quando não tem próximo Bloco de Controle é sinalizado com zero na última posição do Bloco de Controle. Quando um Bloco deixa de ser livre, ele é sinalizado com o zero. Conforme mostra a Figura abaixo:



Essa solução funciona, mas tem um problema que com tempo a Lista de Blocos Livres fica desorganizada. Isso acontece porque os Blocos Livres ficam fora de ordem. E para encontrar um Bloco Livre o SO tem que varrer muitos blocos de controle. Para evitar que isso ocorra não existe uma lista só, o disco (volume) é divido em pedaços e cada pedaço passa ter a sua Lista de Blocos Livres.

Por exemplo, digamos que o volume seja divido em 100 blocos para cada pedaço. Conforme mostra a Figura. Então fica mais fácil achar um Bloco Livre e saber se um Bloco está livre ou não.



Atualmente, essa solução não é mais utilizada, pois o Bitmap é muito mais simples. Mas, antigamente era utilizada pelo UNIX.

VETOR DE BITS (BITMAP) DE BLOCOS LIVRES

O SO usa uma estrutura de controle que agiliza o conhecimento dos blocos que estão livres. Atualmente os SOs utilizam essa solução. Esta lista fica em algum lugar especifico do volume que é chamado de Bloco de Controle Permanente, por exemplo, no ultimo bloco livre. Para saber qual o bloco livre basta olhar para a posição do array. Nesse array estão contidos todos os blocos: com 1 identificamos os blocos ocupados e com 0 os Blocos Livres.

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

1 K Byte = 8 K bits → 1 Bit para 1 Bloco → Então o máximo que cabe nessa estrutura é 8000 Blocos.

Qtd. Entradas em um Bloco = Tamanho do Bloco / Tamanho da Entrada

Tamanho do Bitmap = Qtd. Entradas em um Bloco * Tamanho do Elemento = 8000*1 Bit = 8.000 Bits

Vantagens dos Tipos de Gerência de Blocos Livres

- O Bitmap de Blocos Livres é melhor do que a Lista de Blocos Livres, pois basta ir ao bloco representado pelo index do array e verificar se o bloco está livre ou não.
- No Bitmap o espaço gasto para armazenar o array é sempre fixo em função da quantidade de blocos que o volume tem, por exemplo, se o volume tem 10.000 Blocos o vetor de bits será de 10.000 bits.

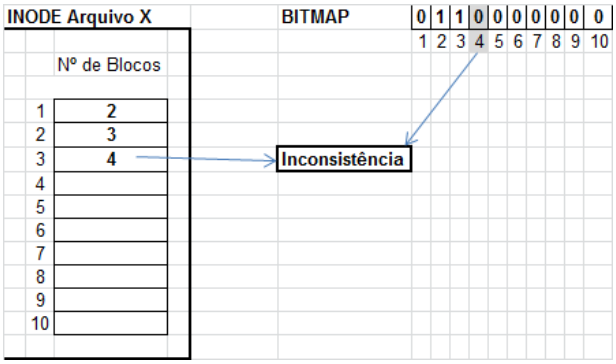
Desvantagens dos Tipos de Gerência de Blocos Livres

- Na Lista de Blocos Livres para o SO descobrir um bloco livre tem que percorrer toda a lista para saber se certo bloco está livre ou não.
- Na Lista de Blocos Livres o espaço gasto para armazena-la varia de acordo com a quantidade de Blocos Livres que o disco tem, por exemplo, se o disco tem poucos blocos livres vai gastar poucos blocos de controle. Se o disco tiver muitos blocos livres vai gastar muitos blocos de controle. Criando uma lista comprida de blocos livres.

Observações:

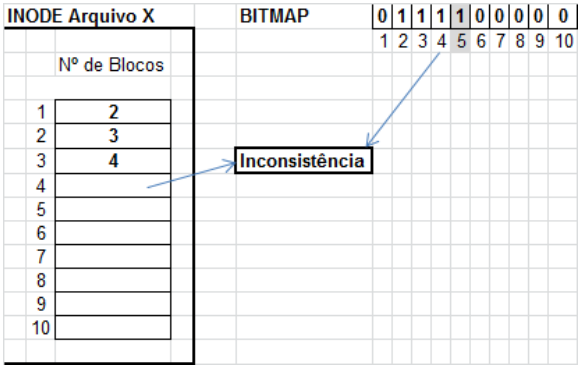
- As estruturas de controle Lista de Blocos Livres e Bitmap são estruturas para aumentar a velocidade na hora do SO verificar se um bloco está livre. Apesar de o Bitmap ser melhor do à Lista. Ambas as soluções são melhores do que saber se um bloco está livre por exclusão.
- A verificação por exclusão é feita verificando se o bloco está sendo utilizando por algum arquivo. Por exemplo, temos dois arquivo A e B e o SO precisa saber se o bloco 9 está livre. Então ele verifica se o arquivo A e o B está com o bloco 9 na sua estrutura de controle de blocos utilizados. Se o bloco 9 não estiver no A nem no B, então ele está livre. Mas essa forma de identificar os blocos livres é muito lenta. Portanto, não é utilizada na pratica.

11) INCONSISTÊNCIAS DE BLOCOS



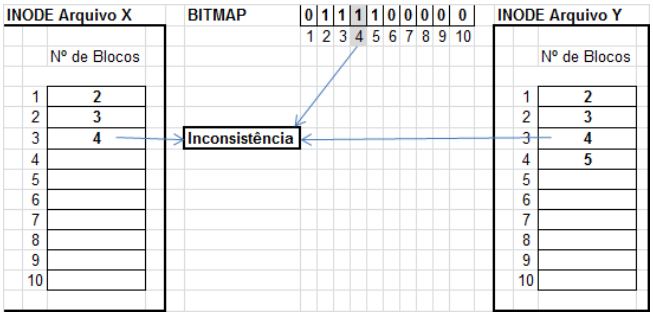
1) O Bloco está marcado como livre na estrutura de controle de blocos livres, mas está presente na estrutura de controle de alocação de um arquivo (Bloco faz parte de um arquivo, mas está marcado como livre)

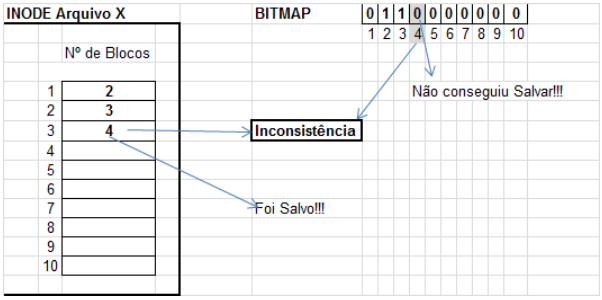
Este tipo de inconsistência pode acontecer na Alocação Encadeada com FAT. Esta é o pior tipo de inconsistência, pois se o bloco está marcado como livre e o sistema operacional vai tentar alocar este bloco a outro arquivo então a informação ficará perdida, pois o SO vai sobrescrever a informação.



2) O Bloco <u>não</u> está marcado como livre na estrutura de controle de blocos livres e não está presente na estrutura de controle de alocação de nenhum arquivo (Bloco está marcado como ocupado e não pertence a nenhum arquivo)

No Windows isso é chamado de Lost Clustr (Bloco Perdido).





3) O Bloco está presente na estrutura de controle de alocação de mais de um arquivo (Bloco faz parte de dois arquivos)

Como são duas estruturas de controle diferentes, sendo uma para controlar os blocos ocupados e a outra para controlar os blocos livres, então ao atualizar uma, deve ser atualizado a outra. No caso de falta de energia no meio da atualização dessas estruturas de controle pode ocorrer este tipo de inconsistência, pois só salvou um pedaço.

Para solucionar este tipo de inconsistência existem duas soluções:

1) Execução automática de um programa que procura inconsistências (Unix:FSCK e Windows:SCANDSK). Quando o SO é reiniciado depois de um desligamento abrupto.

2) Utiliza o conceito de transação (Atomicidade – a atualização do INODE e do Bitmap acontecem juntas ou não acontecem)

12) DESEMPENHO NO USO DE ARQUIVOS

CASO 1 – Leitura de Arquivos

```
Char C;
fd = open("dados.dat", o_RDONLY);
for(i=0;i<1024; i++){
        read(fd, &c, 1)
        //uso a variável 'C' para alguma coisa
}
close(fd);
```

Quantas vezes, no LOOP, o disco é fisicamente lido? Uma única vez, pois essa é a menor unidade de leitura: um bloco. Quando é feita a chamada de leitura o bloco é carregado para a memoria. Os blocos que são lidos são guardados na Cache de Disco (Disk Cache). O bloco é lido por inteiro e é mantido na memória, dentro da área reservada do SO.

CASO 2 – Escrita em Arquivos

```
fd=open("dados2.dat", O_WRONLY)
for(i=0;i<1024;i++){
//faço um calculo e atribuo a variável 'c'
write(fd,&c,1);
close(fd)
```

Quantas vezes no LOOP, o disco é fisicamente escrito? Depende da abordagem de Cache de Disco utilizada pelo SO que são duas:

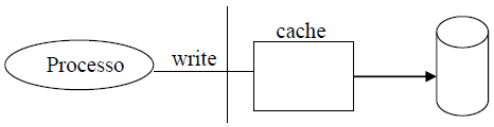
- CACHE WRITE THROUGH = 1024 vezes
- CACHE WRITE BACK = 0 vezes

CACHE WRITE THROUGH

O dado é escrito na cache de disco (que fica na memória) e no disco. Neste tipo de cache a chamada WRITE atravessa a cache e escreve pelo código acima 1024 vezes. Ele escreve na cache e no disco um bloco por vez. No caso de leitura vai ler no disco, ou seja, o disco será acessado uma única vez só se o que for lido couber em um bloco.

O Unix faz uma leitura à frente e já coloca na cache sem que o processo peça. Mas isso só acontece se a leitura for sequencial, pois se for uma leitura randômica não tem vantagem nenhuma.

A cache de disco é uma parte do SO que guarda os blocos recentemente lidos. Desta forma, se for necessário ler o bloco novamente, ele será lido da cache e não do disco que é mais lento.

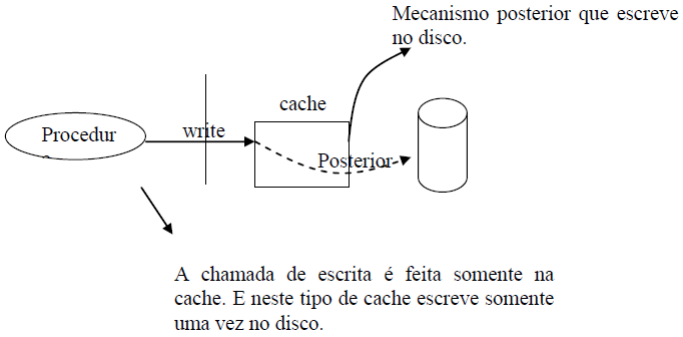


CACHE WRITE BACK

O dado é escrito na cache do disco e depois gravado de uma só vez no disco. Essa atividade de gravação em disco (escrita física) ocorre em duas situações:

1) De tempos em tempos. (Ex.: Unix de 30s em 30s)

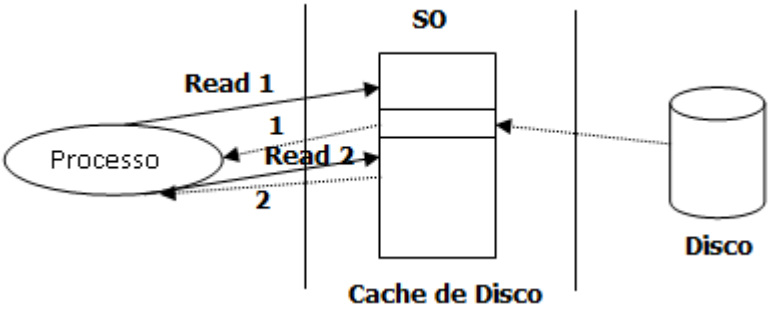
2) O bloco é escolhido como vítima e ainda não tinha sido escrito no disco.



A escolha de bloco vítima é feita utilizando o algoritmo LRU. O bloco no fim da lista é o que vai ser descartado é o bloco vítima. O SO mantém uma lista encadeada dos blocos que estão na cache em disco e mantém a ordem de uso, então o SO sabe quem foi o bloco menos recentemente usado.

Leitura da Cache de Disco

A primeira vez que o bloco é lido é preciso busca-lo no Disco para salva-lo na Cache do Disco. Na segunda vez que o mesmo bloco precisar ser lido não é preciso buscar no Disco, pois ele já se encontra na Cache do Disco. Conforme mostra a Figura abaixo:



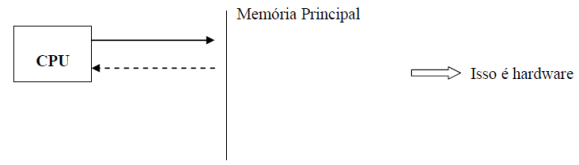
Essa solução é utilizada no Browser. A primeira vez que uma pagina é consultada pelo Browser ela é buscada no servidor e salva na cache do Browser no Disco Local. Quando essa mesma pagina for acessada outra vez a pagina é consultada da cache atualizada no Disco Local. A vantagem desse tipo de solução é que aumenta a velocidade de acesso a pagina web.

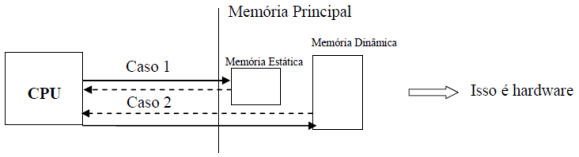
Memória Cache

A memória cache é uma memoria intermediaria. A memória principal é dividida em duas partes: memória estática e memória dinâmica, esta última é mais barata, embora seja mais lenta enquanto a memória estática é mais rápida. A memória estática é usualmente chamada de memória cache. A memória dinâmica existe a necessidade de carregamento dos capacitores para manter o valor do dado armazenado. Tem dois casos:

1) Quando o dado está na memória estática e a CPU pega o dado direto da memória estática: Este caso a CPU não espera pelo dado por ser rápido o acesso.

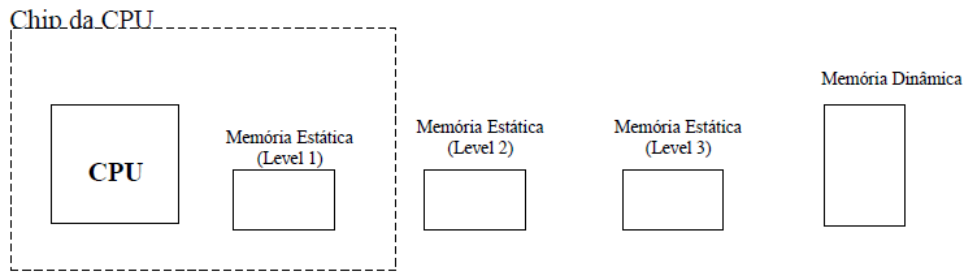
2) Quando o dado não está na memória estática e precisa ir até a memória dinâmica. Neste caso a CPU espera.





Hoje em dia tem vários níveis de memória estática se o dado não estiver no nível 1 vai no nível 2. Se não estiver lá vai para o nível 3 e assim até achar o dado. Sendo que o nível 1 é mais rápido que o 2 e o 2 é mais rápido que o 3 e assim sucessivamente.

Como a memória dinâmica é muito cara, então a solução foi quebrar em vários níveis, solucionando o problema do custo. A cada nível diminui o custo e em compensação ela fica mais lenta.



CACHE DE DISCO E PAGINAÇÃO SOB DEMANDA

Na **cache de disco** tem mais velocidade usando os blocos em memória enquanto na **paginação sob demanda** tem mais capacidade usando o disco. Ou seja, na paginação sob demando usa-se o disco para guarda as páginas vítimas e assim ganhar capacidade em memória. Na cache em disco usa memória para aumentar velocidade de acesso ao disco. Então, um quer usar para ganhar velocidade (cache de disco) e o outro para ganhar capacidade (paginação sob demanda).

Se a memória encher como ficaria a cache em disco? Qual o melhor tamanho para cache em disco? O que se pode fazer quando a memória encher é diminuir a cache em disco então precisamos definir quanto de tamanho pode ser diminuído. Essa competição por memória é um problema. Os sistemas operacionais antigos, Unix, estabelecem um cálculo para diminuir a cache.

Os sistemas operacionais novos (NT, Linux) coloca isto em um único pacote que é chamado de gerência integrada, a gerência da cache e das páginas vítimas.

Vantagens

- A cache de Disco Write Back é muito mais rápida do que a Write Through.

Desvantagens

- A Cache de Disco pode perder informação se a energia acabar abruptamente.

Observações:

- Cache de disco é diferente de cache de memória. A primeira usa a memória para "simular" o acesso ao disco com o propósito de aumentar a velocidade. Já a segunda (Mem. Virtual) usa o disco para "simular" a memória com o motivo de aumentar a capacidade.
- O dado na cache não é bloqueado. A cache é implementada pelo sistema operacional, software.
- A cache de disco é só para arquivo.
- Buffer guarda a informação de forma temporária. E a cache guarda a informação de forma especifica com o objetivo de aumentar o desempenho, a velocidade.