Memória

A memória é como se fosse um Array e cada elemento dentro dela é endereçado com um número começando pelo 0. Além disso, cada elemento possui um byte de tamanho.

Cada endereço é único e para alterá-lo é preciso informar qual é o seu endereço na instrução em **linguagem de máquina**. Essa alteração pode ser de byte em byte, mas como isso é muito pouco, pode-se também alterar em maior quantidade a cada instrução (em potências de 2). Por exemplo, uma instrução de 64 bites pode alterar 8 bytes por vez.

Problema: Programas x Processos (SO1)

O usuário manda executar programas, que **é um arquivo executável com código em linguagem de máquina que fica no disco**. O programa é lido para a memória e se transforma em um processo, que é o programa em execução, ocupa memória.

Só o processo que de fato é útil para alguém, pois a CPU não consegue executar instruções do programa que estão em disco, somente as que estão em memória.

Código em linguagem de máquina

O código que é executado de verdade é em linguagem de máquina, porém ele tem limitações no acesso a memória. Além disso as variáveis **perdem seus nomes**, vão para o endereço de memória e **se usa o endereço de memória**.

Intel Linguagem de máquina - i =i+1;

MOV AH, [2000] -> Copia para o registrador AH o valor contido no endereço 2000 da memória

INT AH -> Incrementa o valor contido no registrador AH
MOV [2000], AH -> Coloca o valor incrementado no endereço 2000

Chamar outro código na memória, que pode ser uma sub-rotina – Calcula();

CALL 500 -> Chama a sub-rotina que está localizada no endereço 500

Problema: Endereçamento

Na mesma memória podemos ter vários processos ocupando endereços diferentes, então como o compilador gera o código de máquina, que usa sempre endereços, se não se sabe a priori onde vai ser carregado o processo na memória? **Como carregar e executar um programa na memória a partir de um endereço que não é previamente conhecido?**

Soluções: 1) Correção de endereços em tempo de carga

Solução simples e que depende do hardware.

Essa solução é usada no MS-DOS (Microsoft) nos programas *.exe*

MOV AH, [2000] -> i

O compilador compila supondo, por exemplo, que o programa vai ser gerado no endereço 0 de memória. Com base na suposição ele vai definindo onde fica cada elemento que o programa utiliza (variável, sub-rotina etc.). Ou seja, o **Hardware** define onde está cada coisa e partir dessa definição ele coloca os valores nos operandos que utilizam essas coisas. Por exemplo, definiu-se que a variável **i** está no endereço 2000, na hora que se compilar o **MOV** ele colocará 2000 no **operando 2**.

Se o programa começar de fato no endereço 0 não ocorrerá problema, mas caso já exista um programa nessa posição ele não vai funcionar, pois ele foi compilado com base no endereço 0. Vamos supor que o programa foi posto no endereço 30000, o MOV não estará na posição 2000 e sim na 32000 entrando assim a correção de endereço em tempo de carga.

Se carrega o programa na memória X, como ele está no arquivo executável no disco, e o **SO em cada operando onde se tem endereço altera o valor que está lá em memória**. Então no exemplo o operando 2 tem o valor 2000, mas ele está errado, então na memória o SO troca para 32000, que é de fato onde está a variável i se o programa começar a executar no 30000.

MOV AH, [2000] -> MOV AH, [32000]

Qualquer instrução que utiliza o endereço tem seu valor corrigido para refletir o lugar exato onde está aquela variável, sub-rotina, que só é conhecida depois que o SO sabe o endereço inicial ao qual o programa ficou na memória. Não é mudado no arquivo executável e sim em memória.

Dificuldade: Saber aonde no código do programa estão as instruções que tem operandos que são endereços, pois eles precisam ser corrigidos.

Solução: É resolvido com compilador dizendo onde estão os operandos. Na hora da geração do arquivo executável o compilador coloca uma informação no arquivo executável além do código, uma tabela que diz onde estão os operandos que precisam ser corrigidos quando o arquivo executável for carregado na memória. Essa tabela é usada pelo SO para fazer a correção.

Essa correção é repetida várias vezes, cada vez que executar o programa, tem um peso maior, mas não muito considerável.

2) Uso do Registrador de Base

Precisa de um Hardware mais poderoso.

Hardwares muito antigos não tem o registrador de base.

Foi usado no MS-DOS nos programas *.com*

Essa solução precisa que o Hardware possua um registrador especial chamado **Registrador de Base**. O registrador não é manipulado diretamente pelo programador em linguagem de máquina como os outros registradores e sim pelo SO.

O valor do endereço de memória que um processo utilizará não é o mesmo que está na instrução em linguagem de máquina. Para qualquer instrução que acesse a memória, o valor contido na instrução é somado ao valor contido no Registrador de Base antes de fazer o acesso a memória, o resultado é o endereço que será consultado.

MOV AH, [2000] -> i

Registrador de Base -> 30000

MOV AH, [2000] -> Soma o valor do RB -> MOV AH, [32000]

Essa solução é parecida com a anterior, mas a diferença é quem e quando faz a soma. **No primeiro caso quem faz a soma é o SO depois de carregar o código na memória, mas antes da execução. No segundo quem faz é o Hardware na hora que executa uma instrução que quer usar o endereço de memória. Além disso, a memória não é alterada, o endereço no exemplo continua a ser 2000.**

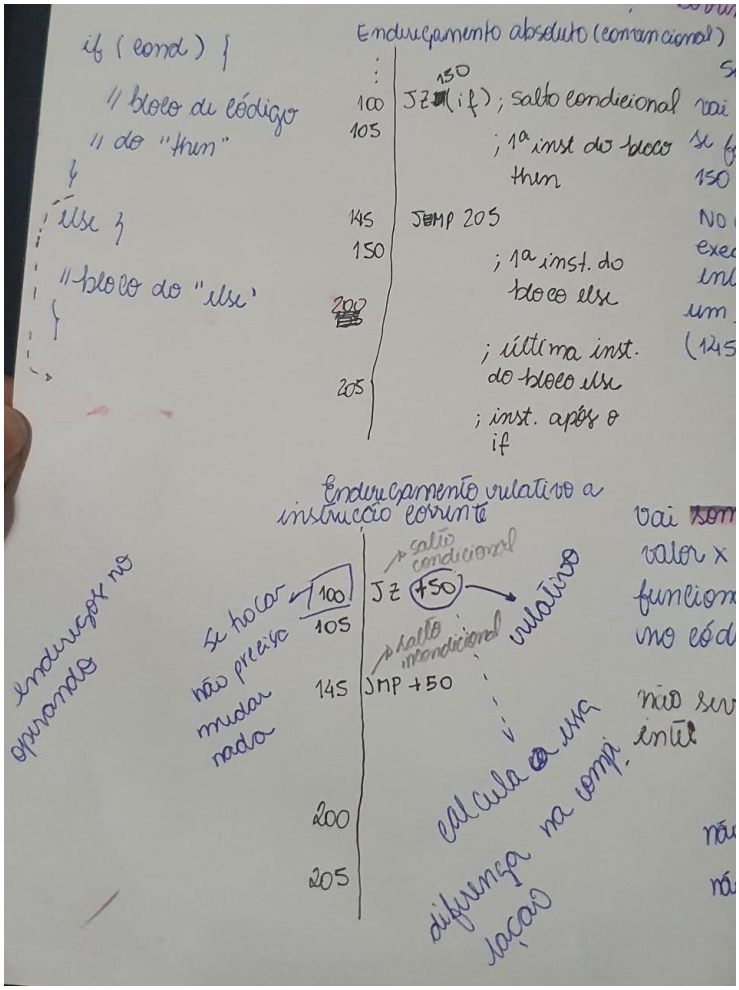
Outra diferença é a quantidade de vezes que a soma é feita. Na correção em tempo de carga é feita essa correção para cada instrução que o código tem uma única vez, já no registrador de base a soma é feita para cada execução, por exemplo, se MOV estiver em um loop para cada execução ocorrerá a soma. No entanto, isso não é ruim porque quem soma é o hardware específico que faz essa soma rápido, então não é um valor significativo e o compilador não precisa fazer nada, não gera tabela.

Quantos registradores de base existem? A mesma quantidade de CPU. Como roda mais de um processo então? Uma máquina com uma CPU só roda um processo por vez, mas vai trocando muito rápido o processo que está executando, dando a ideia de múltiplos processo executando ao mesmo tempo. Importante pontuar que o SO vai trocando o valor do registrador de base conforme os processos trocam.

3) Uso de endereçamento relativo à instrução corrente (Solução Parcial)

Precisa ser usada em conjunto com alguma outra solução.

É feito pelo hardware, mas nem todo hardware faz isso. No caso da **INTEL** tem algumas instruções que usam, como a de salto, mas outras não, como a de MOV.



Em uma condição de if e else utilizando o endereçamento absoluto, ocorre o **salto condicional** se o if não for verdade, ou seja, não é executado a parte “then” (o que está dentro do if). Caso o if for verdadeiro ele precisa pular o else, fazer um **salto incondicional**. Todos esses saltos ocorrem com valores do endereço da memória. Já utilizando o endereçamento relativo à instrução corrente os pulos não ocorrem com endereços e sim somas ao endereço corrente que levam para os endereços, ou seja, utiliza endereços relativos.

Não precisa de correção na instrução, a instrução se auto corrige.

4) Segmentação

5) Paginação

Áreas de Memória de um Processo

A memória possui código executável, que vem do arquivo executável e variáveis (globais, locais e dinâmicas). As **globais** são as mais simples, existem o tempo todo e podem ser consultadas ou alteradas em diferentes partes do código. Já as **locais** só podem ser usadas dentro da rotina onde essa variável foi declarada. As **dinâmicas** são variáveis que precisam de um comando explicito pelo programador para existirem, por exemplo, o comando new para criar uma variável dinâmica.

Local (Pilha) – Página 2 frente

As variáveis locais estão armazenadas na pilha, pois precisam dela para fazer recursão. E são criadas e destruídas durante a execução do programa/rotina.

Devido a recursão se pode ter na memória mais de uma vez a mesma variável local, mas somente uma delas vai estar ativa. A marcação de ativo também marca a diferença entre o que está livre e o que está alocado, tudo que está acima da marcação está em uso ou pode ser usada e tudo que está abaixo já foi usado.

Uma característica da pilha é que existe uma ordem de destruição, a última variável a ser alocada é a primeira a ser destruída (subir a marcação “topo”). Para realizar a alocação e destruição (empilhar e desempilhar) existem as instruções em linguagem de máquina *push*, que empilha, e *pop*, que desempilha. A existência dessas instruções demostra a importância da pilha.

A pilha pode ser mais complicada, basta acrescentar mais variáveis locais na mesma rotina, sendo assim, mais variáveis ativas, não só o topo. Também é guardado na pilha os endereços de retorno, ou seja, quando acaba uma sub-rotina ela precisa voltar a executar a instrução seguinte a chamada. Para isso ser possível utilizamos a instrução call, que empilha o endereço de retorno (endereço seguinte ao call), e ret, que vê onde está o topo da pilha (endereço de retorno), e faz um salto para lá, e esse retorno está apontando para a instrução seguinte a sub-rotina.

Cada pilha possui uma thread.

Variáveis Dinâmicas (Heap) – Página 2 verso

As variáveis dinâmicas normalmente são um registro e precisam ter uma variável ponteiro associado.

De acordo com o exemplo em C, quando se aloca pe1 e pe2 as variáveis vão para a pilha, pois são variáveis do tipo ponteiro e locais. Quando a rotina X começar, ela irá alocar uma variável empregado na heap com endereço inicial 2000, que armazenará a matrícula e nome, sendo assim, pe1 irá guardar o valor 2000. Na alocação do segundo empregado não será possível utilizar o mesmo endereço (2000) e sim 2104, que também armazenará matrícula e nome, sendo assim, p2 irá guardar o valor 2104. Note que sobrou espaço na heap que não está em uso, ele será usado para guardar novas variáveis dinâmicas, a mesma coisa ocorre na pilha, que está abaixo do topo.

Supondo que se deseje destruir as variáveis, então seria dado um free(pe1) e free(pe2), não necessariamente em sequência. No free pe1 ele vai liberar a variável dinâmica apontada pelo pe1, ou seja, ele vai destruí a variável dinâmica que está no endereço 2000, deixando assim a área vazia.

Em linguagens mais sofisticadas, como Java, não é necessário o programador utilizar o free. A própria linguagem vê que não será mais utilizado a variável e a destrói, ou seja, não está mais usando a variável local e apaga a dinâmica. Esse mecanismo se chama **coleção de lixo** e evita que o programador apague algo erroneamente. Porém, ele demora mais tempo para ser executado, mas é melhor para quem está programando, pois ele não precisa se preocupar com isso.

Em linguagens orientadas a objeto (objetos são variáveis dinâmicas) o uso da heap é mais frequente, não só quando se quer fazer árvore e lista encadeada, como podemos ver no exemplo Y. Não há o ponteiro explicitado, mas funciona da mesma forma. Para validar é necessário igualar e2 a e1, na impressão da matrícula irá imprimir o valor de e2 pois são ponteiros e só foi criado um empregado.

Pilha x Heap

Do ponto de vista da memória a diferença entre ambas é a ordem de destruição. A heap não segue a mesma ideia da pilha, a ordem de destruição dela não é conhecida antecipadamente. Já do ponto de vista da programação, não se pode fazer coisas sofisticadas (lista encadeada, árvore) que são feitas na heap na pilha.

Alocação das áreas de um processo juntas ou separadas – Página 3

Nos exemplos anteriores as áreas de código, dados e pilha estão juntas, mas não precisa ser sempre assim. Ao alocá-las separadamente é mais fácil achar espaço, pois é elas ocupam menos espaço do que juntas. No entanto, sua desvantagem é o fato de não ter mais uma base única para cada processo, o que à primeira vista pode inviabilizar o registrador de base. Porém pode haver mais de uma base em cada processo, desde que cada base tenha uma função diferente, então pode se ter a base dos dados, base da pilha e base do código. Isso é possível pois as instruções que são usadas em cada área são diferentes (push/pop, mov e jumps e calls), logo, o hardware sabe qual base usar com base no tipo de instrução (**Intel** permite isso).

Se as áreas podem estar separadas, por que eu preciso de uma área de cada tipo? Quantas áreas de cada tipo podem existir em um processo?

Código e dados não possuem restrições, mas não será possível usar o registrador de base, pois ele precisa de uma base única de código/dado. Porém a pilha tem uma regra clara de quantas áreas podem existir por processo.

Quando utilizamos as instruções de push e pop, incrementamos ou decrementamos a o topo da pilha (registrador Stack Pointer, que é único quando a máquina tem somente uma CPU, mas o SO vai mudando o valor do topo de acordo com a thread que está em execução no momento). Supomos que tenhamos duas pilhas no mesmo processo, o Stack Pointer não vai funcionar, pois ele não iria pular de uma pilha1 para pilha 2, e sim continuar descendo na pilha 1, então uma thread (pontos de execução), que é quando se tem execução simultânea em dois pontos do mesmo código e um processo simples só tem uma, só pode ter uma pilha.

Agora vamos supor que estamos executando duas thread na mesma pilha e em uma única CPU (dá a impressão de execução simultânea, mas é um pouco de cada uma). A rotina começa empilhando uma variável, mas não termina e o SO escalona outro rotina e empilha outra variável em seguida e assim por diante. Quando acaba uma rotina em um thread será necessário desempilhar (subir o topo) e não será possível, pois será necessário desempilhar uma variável que ainda está sendo usada em outra thread. Logo, isso não funciona e é necessário uma segunda pilha para que a alocação e deslocação das variáveis locais ocorram de forma independente em cada thread.

Falta de memória

Ocorre quando a memória do computador é insuficiente para executar o que é desejado.

Soluções: 1) Overlay - Página 4

Muito antigo, ruim e não se usa mais. Fazia sentido quando o código não cabia na memória.

O Overlay consiste em quebrar o código do programa em partes e não as carregar ao mesmo tempo na memória. Por exemplo, se carregava a A e a B e iria carregando por cima as outras partes conforme a necessidade. O problema é que essa solução é que se perde desempenho, gasta-se mais tempo, toda hora é necessário ler as partes do disco. Além disso esse controle é feito no programa e não pelo SO, pois na época ele era muito simples, logo, eram necessários programas mais trabalhosos.

2) Swap de processos - Página 4

Mais sofisticado. Diferente do Overlay, o SO faz as trocas e essa solução sempre funciona.

O Swap ocorre quando o espaço da memória não cabe mais um processo que precisa ser rodado. Para resolver esse problema o SO escolhe alguns processos e os copia para o disco. A sua desvantagem é similar ao do Overlay, o código está em disco não roda, pois, a CPU só consegue executar o que está em memória, logo, para rodar outro processo é necessário trocar.

Outro problema são os endereços, digamos que na troca de um processo que estava na memória, foi para disco e voltou para a memória o endereço mudou, então as variáveis que tinham o primeiro endereço estão apontando para o local errado. Para resolver esse problema é necessário utilizar o **registrador de base**, pois com ele o endereço não conta a partir do 0 da memória e sim da base do processo. Portanto é só o SO mudar a base do processo para refletir onde ele está.

Uso de **endereço absoluto** pode ser um problema, por mais que **correção em tempo de carga** já funcione, o compilador consegue marcar e passar para o SO os operandos que são endereços, mas ele não sabe onde estão os ponteiros.

Para escolher qual processo irá para o disco o SO escolhe os que estão bloqueados, visto que os executando estão executando e os prontos estão na eminência de ser executados. No entanto, nem todo bloqueio são iguais, alguns podem demorar muito (mais de um segundo) e outros poucos, por exemplo bloqueio de ler arquivo, sendo o segundo opções ruins para serem escolhidas.

Para controlar o que está ou não em disco poderia ser criado uma variável, mas o UNIX, que utiliza o swap de processos, optou por criar estados novos, sendo eles: “bloqueado em disco” e “pronto em disco”, que é quando ele sai de bloqueado em disco, pois o que o bloqueava acabou e agora ele precisa voltar para a memória.

Já o Android utiliza uma variação do Swap de processos pois ele possui uma relação de disco e memória bem diferente de um computador, que tem uma diferença entre a memória e disco muito maior, sendo o disco maior. O swap põe tudo em disco, mas como em celular o disco é pequeno ele se encheria muito rápido, então quando a memória enche se escolhe um processo e em vez de salvá-lo em disco, ele avisa para o processo que ele irá morrer e somente é salvo em disco as variáveis importantes. A vantagem é que **ele gasta muito menos disco** salvando as variáveis importantes do que no swap. A desvantagem é que se você quer acessar um processo que já morreu o Android manda criar um processo novo a partir do mesmo programa morto e ele tem a obrigação de ler as variáveis salvas em disco e montar uma tela igual, dando a ideia para o usuário que é o mesmo processo. Essa funcionalidade é programada pelo programador de cada programa, sendo assim podem ocorrer erros no salvamento ou recuperação, o que não ocorreria se fosse feito pelo SO. Isso aqui vai babar para ponteiro, vai precisar fazer algo inovador aí ou salvar em base de dados.

3)Biblioteca Compartilhada (Não usa o Disco e nem tem Registrador de base) – Página 4 verso

Mecanismo que economiza a memória fazendo com que os processos gastem menos memória.

No Unix se chama de biblioteca compartilhada, já o Windows a chama de biblioteca de linkagem dinâmica, (dynamic link library – **DLL**).

Na compilação não se sabe onde estão os endereços das instruções, é necessário fazer a link-edição, que além de juntar os objetos no arquivo executável, preenche as lacunas com os endereços corretos que a instrução precisa utilizar. No entanto, as vezes em vez de dois arquivos fontes, temos uma biblioteca (rotinas úteis) que já foram programadas, por exemplo o printf, e na hora da link-edição o código dessa biblioteca é incorporado ao arquivo executado e assim o link editor consegue preencher a lacuna do código usado. Ou seja, o arquivo executável não tem somente as rotinas que o programador compilou, também possui as rotinas da biblioteca.

O problema da biblioteca tradicional é que se dois processos usarem a mesma biblioteca ela será inserida duas vezes na memória, ou seja, há uma duplicidade de

conteúdo. A solução para esse problema é a biblioteca compartilhada, onde a biblioteca não é inserida no arquivo executável, ela segue separada. Um problema dessa separação é que o link editor não consegue preencher as lacunas de endereço para a biblioteca. O SO precisa fazer o trabalho de link editor para preencher a lacuna, ele precisa descobrir onde estão as rotinas na biblioteca e as preencher (é indicado no arquivo executável onde estão as lacunas e ele preenche), isso é feito um pouco antes da execução, na hora que *a biblioteca (?)* está sendo carregado na memória.

A biblioteca compartilhada não funciona com registrador de base, é necessário utilizar endereçamento absoluto. Além disso ela funciona bem na segmentação, mas não tão bem na paginação.

OBS: A biblioteca só existe enquanto tiver um processo que use ela na memória, quando ele para de existir ela também para

4)Segmentação sob demanda

5)Paginação sob demanda

Fragmentação de memória – Página 3 verso

Fragmentação de memória ocorre quando há espaço livre na memória para alocar um processo, mas ele não é contínuo. Isso ocorre devido a alocação e desalocação dos processos. No entanto, mesmo alocando separadamente o processo vai seguir existindo fragmentação, mas em um grau menor.

Para solucionar esse problema usamos a compactação (“similar ao SWAP de processos”), onde os processos são deslocados para que o espaço da memória que estava fragmentado fique em sequência, podendo assim ser alocar um novo processo. No entanto, ela só é feita quando há um processo querendo ser alocado que não cabe no espaço contínuo da memória.

Para saber qual processo deslocar, o SO, que faz a compactação, possui uma tabela de controle de processos e dentro dela tem um array de mapa de memória que tem a informação de onde está o processo na memória, sendo assim sabe onde tem buracos na memória.

Um problema na compactação, uma variável dinâmica vai trocar de lugar e se não se fizer algo ela vai começar a apontar para o lugar errado. É necessário fazer uma correção de endereço, mas o SO não sabe onde está o ponteiro, então ele não irá conseguir fazer a correção. Para fazer a correção é necessário utilizar o Registrador de Base, mesmo problema do Swap de processo.

Segmentação (Lembra Registrador de Base) – Página 5

A segmentação foi usada na versão inicial do Windows (Windows 3.0) e depois a Intel começou a usar a paginação.

A segmentação apresenta uma alteração na linguagem de máquina, que agora passa a ter dois campos: Número do segmento e deslocamento dentro do segmento.

Call 3:500

Segmentação X Registrador de Base

A diferença entre a segmentação e o Registrador de base é que não é limitado a 3 a quantidade de quebras de áreas e subáreas em um processo. Além disso, a base está no registrador que não vai na memória, sendo assim é mais rápido do que a segmentação que vai na memória. **Sua semelhança é o fato que o endereço usado começa a contar a partir da base.**

A segmentação precisa de um Hardware mais sofisticado para funcionar e da tabela de segmentos (feita pelo SO), que é única para cada processo. A tabela fica armazenada em memória e possui um registro para cada segmento que o processo tem e armazena o início e tamanho de cada segmento.

Durante a execução o Hardware usa os dados dessa tabela para executar a instrução. Primeiro ele olha o número de segmento para identificar qual registro da

tabela vai ser usado, depois verifica se o tamanho do deslocamento é maior que o segmento (o registrador de base da Intel não valida o tamanho da área, se usar vai funcionar e acessar a área de outro processo), se não ele soma o valor do deslocamento ao valor da base.

Caso o deslocamento de segmento seja maior do que o tamanho do segmento é gerado um uma interrupção pelo Hardware, ou seja, a CPU para de executar o código que ela estava executando e passa a executar o código na rotina que trata que o tipo de interrupção, no caso, interrupção de acesso errado a memória. A rotina do SO (sempre trata as interrupções) que trata esse tipo de interrupção tipicamente aborta o processo. Esse controle se chama **proteção de memória**, ele impede que um processo acesse uma área de memória que não o pertence. Quando ela não existe pode gerar uma instabilidade nos processos, pois eles podem ter suas variáveis alteradas sem estarem esperando e assim acarretando um mau funcionamento.

O Hardware não só controla o acesso as áreas, ele também controla o tipo de acesso que é feito as áreas. Quando não existia ou era implementado errado o bit era comum ter vírus que pegavam essa vulnerabilidade, o que ocorreu na Intel. Existe também na tabela as colunas alterável e executável, onde recebem os valores de um bit de verdadeiro ou falso (1 ou 0). Áreas de código e bibliotecas podem ser executáveis, mas não editáveis, já as áreas de pilha e dados podem ser alteráveis, mas não executáveis.

Tipicamente para se alterar uma área de código é necessário alterar o código fonte, recompilar e mandar executar de novo, logo, não faz sentido alterar no meio da execução. Pilha e dados são variáveis, então não fazem sentido ser executadas. Caso tente-se editar um código ou executar uma pilha, o Hardware irá gerar uma interrupção, pois o bit referente está desligado na tabela de segmentos do processo, que irá tipicamente abortar o processo.

Importante pontuar que nem todo processo que sofre uma interrupção é abortado, pode ser que quando ele tenha acessado a memória de forma errado ele estivesse fazendo algo importante e se abortar tudo será perdido. Então cada processo pode pedir ao SO que ele não seja abortado caso ele cometa um erro de acesso a memória, ele solicita que seja executado uma rotina e após isso é abortado o processo. Um exemplo real é o Word, ele não tem garantia que não há bugs e se ele acessar a memória de forma errada e ele será abortado. Caso haja um texto que não foi salvo ele será perdido, porém agora existe o auto salvamento e o Word irá pedir para o SO não ser abortado ao acessar errado a memória. O SO executará uma rotina antes de abortar, que irá tentar encontrar na memória o texto que estava sendo editando e vai tentar salvar ele em outro arquivo. Antes do Word morrer ele manda executar um outro word e o antigo morre e novo vê se tem esse arquivo que são salvos pelo irmão que morreu e o novo tenta recuperar esse arquivo. Ou seja, o Word antes de morrer salva na força bruta e outro word tenta recuperar.

Esses erros de acesso a memória não costumam ocorrer porque são muito grosseiros, normalmente eles ocorrem ao usar ponteiro, pois o ponteiro pode estar com um valor errado.

A tabela de segmento é única para cada processo, e o Hardware deve usar a correta na execução de cada processo. **Mas como o Hardware sabe qual é a que deve ser usada?** Existe um registrador que guarda o endereço da tabela de segmento ativa, que é controlado pelo SO. Quando acaba o tempo de execução do processo o SO faz o registrador apontar para a próxima tabela segmento. Se só tiver uma CPU só terá uma tabela que estará em uso em um estante de tempo, que é a apontada pelo registrador.

Por que não podem ser registradores e sim algo em memória? Porque pode haver tabelas com dezenas ou centenas de segmento, o que terminaria em vários registradores e eles são complexos de se construir, então não faz sentido por eles.

Com segmentação x Sem segmentação – Página 5 verso

O fato da tabela de segmento estar na memória faz com que haja um acesso a mais na memória. Por exemplo, em um caso de CALL, se não houver a segmentação a CPU executa a instrução apontada pelo endereço no CALL. Já com segmentação é mais complicado, pois para chegar no endereço final (o que será executado) é necessário o Hardware acessar o segmento que está na tabela de segmento que está em memória, sendo assim acrescenta um acesso a memória, para fazer os paços 2 e 3 (validação e soma) e aí sim acessar o registro final, gerando uma perda de desempenho.

Para solucionar essa perda de desempenho ao acessar a tabela de segmento na memória a Intel criou o **registrador de segmento**. Quando um segmento vai ser usado várias vezes é necessário, em linguagem de máquina, carregar o registrador de segmento (os registradores disponíveis são: DS, CS, SS, (dados, código, stack (pilha)),ES, FS.) com os dados do segmento. Por exemplo, em uma troca de variável, sendo DS um registrador de segmento, é usado MOV [DS : 100], AH ao invés de MOV [1:100],AH (**o Mov é obrigado usar esse registrador).**

MOV DS, 1 -> Carrega no registrador DS (Registrador de Segmento) todos os campos do registro 1

MOV AH, [DS; 150] -> Copiando o conteúdo que está no deslocamento 150 para o registrador AH

MOV [DS : 100], AH -> Copiando o que está no registrador no deslocamento 100

A vantagem ser feito assim é que a tabela de segmento não é acessada além do momento em que se faz a carga, logo não tem perda de desempenho. No entanto há perda de complexidade, agora é necessário se preocupar com a linguagem de máquina, que em maioria das vezes é problema do compilador e não do programador, ao menos que ele esteja programando nela.

Pode ocorrer compactação na segmentação desde que o SO também altere o segmento na tabela, a coluna início.

Todo processo tem na tabela de processo um campo que é o ponteiro que guarda do endereço da tabela de segmento do processo. Então quando o SO escalona o processo ele pega esse campo e copiar para o registrador que aponta para a tabela de segmento ativo.

PAGINAÇÃO - Página 6

Quando as máquinas rodavam um processo por vez tudo era mais simples. Todos os processos começam do endereço 0, ou seja, não precisavam de correção, não havia problema de compactação, pois eles não fragmentavam. O único problema era que o usuário fica triste, pois só podia rodar um processo ☹

(**Partição de memória**) Agora supomos que temos uma memória de 4GB e separamos ela em 4, cada uma com 1GB. Manteríamos todos os processos começando no endereço 0, poderíamos executar mais de um processo, porém o tamanho máximo de um processo é de 1GB, ocorreria desperdício de memória (sobra sempre um cadinho em cada memória) e somente poderia ser executado 4 processos por vez, ou seja, limitaria a quantidade máxima de processos que poderia ter na memória.

A paginação almeja o mundo simples, onde todos os processos começam no endereço 0 (todos possuem uma memória individual), mas ao mesmo tempo não há limite de tamanho ou quantidade de processos que podem existir e com pouco desperdício de memória.

Na paginação cada processo vê a memória de uma forma diferente do que ela de fato é. Ele vê as páginas em uma outra ordem, a ordem da memória lógica (que não é real) e acha que sua memória lógica é a verdadeira. As páginas sem conteúdo dessa memória não existem no mundo verdadeiro. Quem faz a interferência no endereço de memória que um processo está usando, ou seja, ele muda o endereço, é o Hardware.

Além disso, na paginação cada processo acha que está sozinho na sua memória falsa, mas tem uma memória que de fato é verdadeira que junta vários

processos. A vantagem é não ter que fazer correção em tempo de carga (não precisa usar registrador de base) porque todos os processos começam a partir do endereço 0. Não tem problema de fragmentação, porque todo mundo está sozinho na memória e não tem desperdício porque as páginas vazias na memória lógica não existem de verdade. Já na memória física não tem desperdício, pode-se alocar a todos os gigas da máquina com processo.

Conversão de endereço (Endereço lógico para físico)

MOV AH, [8200] (Endereço lógico) -> conversão para endereço físico (verdadeiro) -> Endereço na memória verdadeira

Para o Hardware fazer a conversão par ao endereço verdadeiro, ele precisa da tabela de página, que possui no registro de cada página falsa que ele tem. Ela é criada pelo SO onde se diz aonde em cada página do mundo falso está no mundo verdadeiro e marca as páginas do mundo falso que não tem conteúdo.

Como funciona a paginação (Hardware)

1- É necessário quebrar o endereço lógico em duas partes, o número da página lógica e o deslocamento dentro da página e para chegar nesses números é uma conta:

Número da página lógica = int (endereço lógico/tamanho da página)

Deslocamento dentro da página = endereço lógico %(resto) tamanho da página

MOV AH, [8200]

Int (8200/4096*) = 2 *Tamanho da página pela intel é 4096

8200%4096=8

2- Em seguida é preciso converter o número da página lógica em número da página física. Para isso o Hardware irá usar a tabela de página, pois ela guarda a correspondência entre a página lógica e a página física. Aqui também ocorre validação do bit válido, como o passo 2 da segmentação. Ou seja, se o Hardware tentar acessar uma página lógica que não está com o bit valido ligado ocorre uma interrupção.

Além disso há a validação dos outros bits (executável e alterável), e a validação é similar da segmentação. Não se pode tentar executar uma página que está com o bit executável 0 (desligado) e nem alterar uma página com o bit desligado. E caso ocorra será gerado uma interrupção e o SO tipicamente aborta o processo. Da mesma forma que a segmentação, também há um mecanismo que permite que o processo peça para não ser abortado quando ele comete um erro e indicar uma rotina quando ocorre o erro e somente depois dela o SO aborta o processo.

Pela tabela a página falsa 2 se transforma em página verdadeira 0

3- Por fim é feito a recomposição do endereço físico

Endereço físico = número da página física * tamanho da página + deslocamento

0*4096+8 = 8 (Endereço da memória física)

Bits de controle da tabela de página

1 Válido

2 Executável (igual da segmentação)

3 Alterável (igual da segmentação)

Devido aos bits de controle existe uma lógica para organizar as áreas. A área de dados não pode começar em seguida a área de código na mesma página, pois os bits de controle executável e alterável iriam estar ligados ao mesmo tempo, gerando um uso errado. O código poderia ser alterado e a parte de dados ser executada. Então para corrigir isso, fica um buraco sem uso na página e a próxima área começa na próxima página, isso gera uma tabela de página preenchida corretamente e não há uso indevido. A parte negativa é que irá ocorrer o **fragmento interno (fragmentação interna)**, que está contida no processo alocado, mas sem uso. Esse fragmento é um

desperdício, mas inevitável, a menos que a parte ocupe a página toda. Mas não chega a ser grave, pois, a página não é grande.

Perda média devido a fragmentação interna: (pior perda + melhor perda) /2

A melhor perda é 0, quando o tamanho da área é um múltiplo exato do tamanho da página. O Pior caso é quando se tem uma área de código que ocupa um byte a mais em uma página (na intel é 4bytes -1)

Perda média = 0+4095/2 = 2047,5

Desvantagem da paginação – Página 7 verso

A paginação permite crescer as áreas, o que não ocorre na segmentação, pois precisa compactar a área para crescê-la. A área de pilha possui um tamanho médio, onde se conclui que cabe tudo que precisa, mas pode ocorrer dela estourar (o topo da pilha alcançar o limite inferior dela, pois criou muita coisa ou muita recursão) e isso gera uma interrupção e o SO tem que tentar aumentar o tamanho da pilha para resolver o problema. Na segmentação isso é mais difícil porque as áreas estão alocadas na memória física, então é fácil aumentar quando a área vizinha da pilha está vazia, mas em outro caso é necessário realocar a pilha para outra parte da memória que tenha mais espaço para ela crescer. Na paginação o esforço é mínimo, basta encontrar uma página física livre e falar que ela vai guardar outra página que lógica que vai ser pilha.

A área de dados crescem para cima quando a heap fica cheia, então é importante ter espaço entre a área de dados e pilha. Quando a heap fica cheia (ocorre quando o computador aloca uma variável dinâmica e é controlado pela biblioteca da linguagem) existe uma chamada ao SO da parte da biblioteca que aumenta o tamanho da área de dados.

Processos com paginação tem só uma área de pilha e o resto é alocado aos poucos.

Bits invertidos

Os bits alterável e executável são normalmente invertidos, quando um é 0 o outro é 1. A exceção é em linguagens mais recentes, como java. Na compilação é gerado o arquivo compilável e depois a link edição que gera o executável, mas em java não é assim. O compilador não gera a linguagem de máquina e sim uma linguagem intermediária, que é próxima a linguagem de máquina, mas não é ela. Em java se chama bytecode, que o hardware não executa as instruções e sim o interpretador (software) que faz o papel de CPU, logo, mais lento. Java.exe é o interpretador java/máquina virtual java.

O arquivo compilado java não vai para área de código, e sim o java.exe, que é o arquivo em linguagem de máquina do interpretador, já a .call vai para área de dados. Por demorar muito a atualmente editaram a máquina virtual java para fazer uma compilação durante a execução (just in time compiler - evita de ter a interpretação várias vezes), que compila o código bytecode para a linguagem de máquina e faz isso método a método, fazendo eles serem mais rápidos, pois os métodos que rodam mais são em linguagem de máquina.

Na terceira vez que executa ele fica rápido porque está sendo executado indiretamente em linguagem de máquina.

Quando compila o bytecode para linguagem de máquina tem que ser guardado na memória, quando ele vai para a tabela de página ele tem que ser alterável pois quer por novos conteúdos (código compilado) e por outro lado ele precisa ser executável o conteúdo dos métodos em linguagens de máquina. Então ele fica com 1 em 2 bites. Isso também vale para a segmentação (isso fica além das 3 áreas normais)

Melhoria dos paços 1 e 3 da paginação (quebrar endereço em 2 partes e gerar endereço físico a partir da página física) – Página 8

Na segmentação tem soma e comparação, que são simples comparado com paginação, que são divisão e multiplicação. No Entanto na base 2 é fácil, se faz um corte, então se livra da parte 1 e a multiplicação é tirar o corte

Existe um problema na paginação que equivale ao problema da segmentação de desempenho. Pro paço 2 (converter a página lógica em página física) executar é necessário consultar um dado na tabela de página, que está em memória. Então é necessário um acesso a memória para fazer a paginação para pegar esses dados e isso perde desempenho. Na paginação isso é resolvido com registrador de segmento, onde se carrega os dados do segmento que você usa mais, porém a paginação não pode usar isso, **pois paginação é transparente** para o processo, o código do processo em linguagem de máquina não sabe que tá gerando em um computador com paginação. Um programa de 25 anos atrás em uma máquina sem paginação e se por para rodar com uma máquina com paginação ele vai funcionar, porque ele não sabe que tem paginação.

A segmentação é visível pois na linguagem de máquina passa a ter 2 campos.

Então é usado um elemento de hardware, a **TLB (translation look aside buffer -similar ao LRU / tabela resumo),** que é o resumo na tabela de páginas (não tem o bit válido) que seria equivalente a um registrador. Nela estão os registros mais usados (hardware escolhe).

A diferença é que não tem o bit válido e tem o campo o número da página lógica, que na tabela normal seria o equivalente ao índice da tabela de página. Não tem o bit válido, pois seria um desperdício colocar elementos com o bit valido 0, pois acessá-la irá gerar uma interrupção e abortar o processo, logo, não vão para a TLB.

Para cada processo novo que entra em execução a TLB é limpa e colocado os novos valores.

(página 8 verso tem as contas)

Na máquina sem paginação, qualquer acesso gera 100 ns, pois só tem memória física. Já o com paginação é 120 ns, no exemplo ele teria um tempo de acesso a memória maior em 20%. Uma perda de conversão, essa perda é menor se a conversão foi por TLB e maior se em menos casos foram na TLB. Mas mesmo com TLB ela é um pouco mais lenta -> perda de desempenho. Mas a paginação compensa isso nos seus benefícios.

Algumas TLB tem um campo a mais, o número da memória lógica. O hardware consegue diferenciar entre os processos com a existência desse campo. Com esse campo ele faz a TLB não precisar ser esvaziada toda vez que troca de processo em execução, pois com esse campo ele sabe de quem é o registro e se pode usar ele ou não. Ele só usa os registros do processo em execução. O campo permite que quando o processo volte a executar o hardware ainda encontre registros dele e faça com que ele seja mais rápido.

A TLB solução do problema de desempenho da paginação.

Espaço gasto com a tabela de página – Página 9

Mesmo um processo pequeno e com poucas páginas, a memória lógica vai continuar com um espaço grande, 4gb, o que não é problema porque elas estão vazias e não estão na física. O problema é a tabela de página, por mais que não gaste memória física, elas precisam guardar essa informação na tabela de página falando que são inválidas.

Solução

Tabela de página em níveis – Intel solução antiga – Página 9

A ideia é quebrar a tabela de página em pedaços e colocar em uma tabela extra, chamada **diretório de página**, que vai ter um registro para cada pedaço da

tabela de página (existe um índice apontando para cada pedaço), o que faz gastar mais memória. A grande jogada é que os pedaços que não têm página válida não existem mais e a entrada no diretório de página fica com endereço 0. Com isso o gasto da tabela de página diminui.

Na Intel a tabela de página é quebrada em vários pedaços (1024), o diretório de página tem 1024 endereços. Cada pedaço da tabela de página vai ter 1024 entradas e com 4kb. Então o gasto para um processo com poucas páginas caem para 12kbm. O problema dessa solução é o desempenho, agora há 2 acessos a memória, um para o diretório e outra para a tabela, mas não percorre a tabela toda. Porém na maior parte dos acessos é por TLB, que não tem perda de desempenho.

Como ele vai no lugar certinho? O endereço lógico

Essa solução não funciona aqui porque as tabelas ficam muito grande. Máquina com 64 bytes, o corte do endereçamento lógico fica muito grande. Os registros dentro da tabela de página vai ser 2^26, o diretório seria muito grande. Diretório e pedaço grande, não funciona para 64 bites.

Fragmentação interna -> perda de meia página por área

Não ter registrador de página -> computador não sabe que tem paginação, como vai ter isso?

TLB> Não tem bit válido e tem o referente ao índice. TLB hardware põem as páginas mais usadas

Quebra do endereço lógico em duas partes é simples pois o tamanho da tabela de página é potência de 2 -> corte

Sempre tá na TLB as páginas, se não tá ele leva

Solução para os 64 bytes – Página 9 verso

Quebrar a tabela em mais subníveis, adicionar níveis intermediários. O Diretório aponta para uma tabela intermediaria que guarda somente endereços, que também não aponta para a tabela de página completa, ela aponta para outra tabela intermediaria e essa tabela intermediaria aponta para a tabela de página. Isso reduz o tamanho da tabela de página e do diretório. > Árvore e folhas

Intel: Agora o endereçamento lógico está quebrado em 4 partes.

A soma do caderno dos bytes da 48 e sobram 16 que tem sempre o valor 0 e com isso perde um pouco do tamanho da memória lógica máxima que um processo pode ter, mesmo assim da muita coisa (tá com 2^48)

O problema é que se faz mais acessos a memória ainda (4 acessos).

Tabela de páginas invertida – Índice é invertido com o número de páginas físicas – Página 10

Tabela invertida é única para todos os processos. E é adicionado o campo próximo da lista.

Vantagem: Tabela única, por mais que possa ser grande é única.

Na conversão da página lógica para física é preciso descobrir onde está a página lógica X do processo Y. Na tabela normal é trivial, vai na tabela de página e olha e um único acesso a memória. Na tabela invertida não é trivial, o dado que não tenho é o índice, uma solução seria ir olhar cada linha, o que podem levar vários acessos a memória, dependendo unicamente da posição. Banco de dados também tem esse problema e se usa uma árvore b para indexar, mas isso é muito complexo para ser uma solução de hardware. O hardware usa a tabela hash, que é associada a função hash, que leva em conta os campos que você quer indexar e o número de processo). Com a tabela hash há o cálculo da função e ir à tabela (um acesso), foi na tabela (um acesso) para confirmar, logo, foram 2 acessos a memória.

No entanto, com 2 processos não é tão simples, pois na tabela o registro do segundo processo pode já estar preenchido (colisão). Para resolver esse problema é

adicionado um campo a mais na tabela invertida que é o ponteiro para o registro no qual teve colisão. -> 3 acessos a memória

Isso não é simples, é mais complicado, mas a vantagem é a menor quantidade de acesso a memória

Idealmente não tem tantas colisões.

Segundo acesso a tabela invertida é mais para conferir (sem colisão).

Quanto maior for a tabela hash menor chance de ter colisão > O mínimo é o mesmo tamanho da invertida.

Em caso de áreas compartilhadas (biblioteca compartilhada) possuem páginas físicas que pertence a mais de um processo, mas a biblioteca invertida mostrada não cabe isso o que faz a real ainda mais complexa.

SO preenche a tabela invertida e a tabela hash (sendo essa quando começa o processo).

Ops2: há campo para saber se a página tá em uso na tabela física

Ops: tabela de página é muito grande, então não é usada

Sistema Operacional na paginação - Página 11

Uma possibilidade seria o SO ter sua própria memória lógica, mas essa possibilidade possui um problema quando ela for implementada com hardware com TLB mais simples, ou seja, sem número do processo. Pois quando troca o processo ativo o hardware esvazia a TLB, então se um processo fizer uma chamada ao SO, ele vai entrar na TLB e o SO sair. Ou então o SO vai executar numa TLB vazia, e em hardware vai ser mais lento por isso. Ai quando acabar o processo o SO. Pontuando que o processo chamar/fazer interrupção o SO é bem mais comum do que troca de processo. > Sempre limpar a TLB

Caso a TLB seja mais sofisticada é possível, pois não esvazia a TLB sempre. Intel não usa essa solução porque antes a ela não tinha o campo que impede que ela seja limpa em cada troca de memória lógica ativa.

(Página 11 verso) A segunda possibilidade é mais sofisticada e a Intel tipicamente usa. Na memória lógica ter também a parte do SO. No entanto, cada processo vai ter o SO nele. Isso significa que está duplicado? Pois ele se comporta como uma DLL (biblioteca compartilhada), as páginas que estão o SO são as mesmas na memória física, então não tem desperdício de memória física. > Não esvazia a TLB porque o SO está na mesma memória.

A desvantagem é que se passa a gastar um pedaço da memória lógica com páginas do SO, mas para 64 bits não faz muita diferença porque as páginas são enormes. Mas para 32 bites é meio chato. Outro problema é que quando você põe o SO na mesma memória lógica do processo o que impede o código do processo enquanto ele está em execução alterar uma variável de dados do SO? O processo pode alterar as variáveis que podem ser alteradas do SO. Isso se resolve com um controle adicional da tabela de páginas, o **campo núcleo**, que fica ligado nas páginas válidas do SO. Mas como a CPU sabe se o código que ela está rodando agora é do SO ou do processo? Ela sabe por que existe o modo de operação, nas CPUs tem usualmente o **modo usuário (não privilegiado) e o modo sistema/núcleo (privilegiado)**.

Os processos normais rodam no modo usuário, quando se faz uma chamada ao SO a CPU passa a executar no módulo núcleo, onde podem fazer algumas coisas, como acessar as páginas com o bit núcleo ligado. Caso tene acessar como usuário o hardware vai gerar interrupção e o SO vai tratar e abortar o processo.

No módulo núcleo pode acontecer várias coisas, como acessar a página como dito anteriormente, alterar o valor do registrador que guarda o endereço da tabela de páginas ativa, comunicação com dispositivos de e/s com certos comandos (in out) só pode ser nesse modo.

O mecanismo de gerar interrupções conforme o modo que você está executando permite fazer uma implementação de máquinas virtuais. Máquina virtual é um software que faz com que consiga rodar no mesmo hardware SOs diferentes e cada SO acha que está executando na máquina física de verdade, o que não está e sim na Máquina Virtual. O software consegue fazer isso. VM vem usado a anos para facilitar administração de servidores.

O SO que roda dentro de uma VM roda no modo usuário, pouca coisa esse SO pode fazer, mas quando esse SO tenta fazer e/s o hardware vai gerar uma interrupção que um SO normal abortaria, mas o controlador de máquina virtual em vez de abortar, ele faz emulação da instrução em SO e não em hardware. Ele consegue tratar a interrupção e devolver para o SO que executou a e/s o dado que o SO obteria se estivesse executando aquela instrução na máquina física.

Emulação – finge que é um hardware que não é, é SO. Se perde desempenho porque SO é mais lendo que hardware, mas isso não é muito porque a maiores operações que o SO roda não são privilegiadas.

Nota: as tabelas são menores que a memória, tá bom? Tá bom

Paginação com uso do Disco – Página 11 verso

Quando não há página física livre, o que fazer quando, por exemplo, a pilha de um processo crescer? O SO irá usar disco (lembra swap de processo), quando a memória fica cheia e você precisa de mais memória física, o SO escolhe uma das páginas que estão na memória física e sobe em disco. E a tabela de página põem um 0 onde estava ela no campo valido. Então o espaço fica temporariamente livre e com isso o SO pode por outra coisa lá, como tratar o estouro de pilha/ o crescimento de pilha de processo.

O que vai acontecer quando esse processo que está com a página lógica em disco tentar usar a página? A página está invalida, o hw gera uma interrupção devido ao acesso a uma página inválida (ele não vê diferença entre os inválidos), a rotina do SO precisa saber diferenciar entre uma página lógica inválida com e sem conteúdo. No caso da página com conteúdo e em disco ele não pode abortar, na rotina do so verifica que a página lógica pertence ao processo e está atualmente em disco e precisa disparar o mecanismo para trazer a página de volta para a memória física. Se a mem ainda estiver cheia, então o so precisa escolher outra página para ir para o disco e ai sim trazer a página lógica que o processo quer usar. Quando a página está de volta de fato o SO termina o tratamento da interrupção e com isso a rotina que gerou a interrupção é reexecutada e a instrução é executada, pois a página está de volta na memória. -> Também chamado de **swap de página.**

Swap de processo x Swap de página – Página 12 verso

Swap de processo salva todo o conteúdo do processo em disco e isso demora tanto pra por quanto para tirar (Android). Também chega a trocar o status do processo. Já na paginação não é o processo todo, só uma página, fazendo com que seja mais rápido.

Outro ponto no swap de processo ele muda de estado e para de rodar pq o processo todo tá em disco, já na paginação é só uma parte do processo que está em disco o resto está em memória e podem seguir sendo utilizadas pelo processo. Então enquanto o processo estiver usando as páginas dele que estão em memória ele continua rodando. Processo segue rodando- não para. Só para se tentar usar uma página em disco.

Uso de disco atrapalha o desempenho, porque ele é mais lento que é a memória, mas funciona!

Hw que percebe e o SO faz ?

Contra: Gasta mais tempo (paginação usando pisco). Cria problema de desempenho, levar pro disco e depois ler gasta tempo.

Se com muita frequência tiver que ir pegar página em disco a perda é enorme, pq buscar a página em disco é 100mil vezes mais lento.

Coisas para evitar usar o disco nos casos que ele não é muito necessário (“melhorias “na paginação usando o disco) – Página 12 verso

1 Quando a página vítima for de código, não a salvar em disco, pois se ela já foi vítima anteriormente e ninguém apagou, ela já está em disco, então não precisa salvar e página de código não é alterável, então não mudou o conteúdo. Ela não precisa ir para disco, é só apagar e disponibilizar a página física que ela ocupava. (também ser para dll) > Isso só funciona se não apagarem o arquivo executável. > SO não deixa apagar o arquivo executável que está que é base de um processo que está em memória. UNIX finge que apaga, mas deixa salvo em outro lugar e só apaga quando ele sai da memória > Treta com arquivo executável na memória e não ele ter sido anteriormente vítima > Arquivo executável já está em disco e de lá sai as páginas de código caso tenha sido vítima e precise voltar > Economiza um acesso ao disco, não precisa salvar a página vítima

2 Se a página vítima foi uma página de variáveis (dados ou pilha) que já foi vítima anteriormente ai pode ser que não precise salvar. Se a página não tiver alteração de conteúdo e ela foi vítima de novo ela não precisa ser salva em disco porque não ocorreu alteração de conteúdo. Se tiver sido alterada vai precisar salvar de novo. Como sei que foi alterada? Precisa de um mecanismo que sinalize para o SO se foi ou não, porque comprar não vale a pena, gasta tempo. Isso é feito com um bit a mais na tabela de página que é o Dirty ou alterado. Quando ele volta para a memória física o SO põem o valor 0 e quando a página sofre uma alteração de conteúdo o hw automaticamente liga o bit. > Hw altera, os outros ele só consulta. Só altera na TLB, antes de sobrescrever na TLB tem que alterar em memória. ´´E meio complexo, mas a Intel faz.

<u>Quando a tabela de página não tem esse bit</u>

Existe uma tabela de páginas auxiliar usada somente pelo SO) hardware não sabe que ela existe), essa tabela tem o bit alterável. Se o hw não conhece como ele altera? O SO mente para o hw e diz que a página não é alterável e muda o valor do bit na tabela de página, mas na auxiliar ele põem o valor correto e o valor alterável 0 (isso quando ela foi vítima e voltou). Quando tentarem alterar um valor nessa página o hw vai na tabela de página, vê que ela não pode ser alterável, gera interrupção, SO vai na tabela auxiliar onde o SO verifica se é mentira ou não então a interrupção foi porque ele mentiu e nessa hora da interrupção e o SO coloca na tabela o valor e termina a interrupção e a instrução é reiniciada e guarda na auxiliar informando que ela sofreu alteração, se o bit está ligado, e se foi vitima de novo ele vai saber que precisa salvar de novo a página >Army e celulares

Por mais que a situação de ser vítima sem alteração seja raro, os poucos que ocorrem justificam essas implementações.

3 Não carregar todas as páginas de código na memória física quando o processo é iniciado, carregar as páginas de código somente quando elas são utilizadas (Carregamento de código sob demanda)

Na tabela de páginas os bits válidos ficam com 0. Ai quando o SO manda executar o método principal vai dar uma página 0/página inválida (código) e vai gerar uma interrupção e o próprio SO vai tratar e ver que não é um erro do processo e o SO carrega essa página (arquivo executável tá no disco). O SO encontra uma página física livre e carrega a página lógica nela do disco e torna a página válida e assim vai pondo as páginas de código. Isso gera uma melhoria, se o arquivo executável foi muito grande e numa máquina sem paginação teria que esperar muito pra carregar. Além disso, um programa grande tem várias funcionalidades e provavelmente não se uso todo o código ao usá-lo, tipo o word, não se usa todas as funcionalidades sempre quando você executa tipo editor de figura geométrica, sem paginação isso seria carregado e ai se gastaria memória com isso, com paginação não > há melhoria de tempo e memória física. Em programas com interface com o usuário não precisa carregar tudo para o usuário executar, podem ir carregando aos poucos, começando pela primeira página e ir usando.

Essa solução tem uma característica de discos magnéticos que é mais rápido ler de uma vez 10mb para a memória do que ir lendo aos poucos 1mb. Mas se você não precisar ler tudo compensa.

Windows tem uma otimizada em carregar primeiro o que mais usa

4 Similar a anterior, mas para dados. Não alocar na memória física todas as páginas de dados quando o processo começa. Alocar a página lógica na memória física somente a primeira vez que é utilizada. >zera o conteúdo da página antes de alocar. Não tem muito benefício porque tipicamente não vem do disco, não é benefício de tempo e sim de memória física > alocar área de dado sob demanda

5 gastar tempo em momento diferente. Quando está trazendo uma vítima nova, mas não tem espaço é necessário escolher outra página. Pra a página voltar a funcionar são 2 acessos, salvar a vitima e ler a pagina vitima de volta pra memória. A proposta é o SO não deixar a memória física ficar cheia, quando ela tá quase cheia o SO começa um processo de escolher vítimas quando ainda não é necessário, ele salva em disco se necessário e desaloca da memória física. Se algum processo precisar de uma que está em disco ele não precisa esperar os 2 acessos, ele só precisa ler e gastar um único acesso o outro foi antes. O importante de lembrar é que esse trem que o SO faz é quando o disco está sem uso porque aí não prejudica ninguém.

O SO mantém sempre uma quantidade de páginas físicas livres (pull de páginas físicas/pull de páginas). Isso é feito quando a memória física está quase cheia o SO faz uma escolha adiantada das vítimas.

Algoritmos de escolhas de página vítima

Página lógica para entrar gera interrupção caso ela não esteja lá

Algoritmo da fila (FIFO) – Página 14

A vítima é sempre a primeira da fila

É ruim, é usado quando mais nenhum outro pode ser usado

Ele escolhe várias vezes páginas vítimas que seriam em seguida usado

Algoritmo teórico – Algoritmo ótimo – Página 14

Teórico porque não dá pra conhecer o futuro

Não existe algoritmo que gera menos interrupções que esse, não tem melhor que ele

Primeira vez que não sei qual vai usar, mas o hw ou processador poderia ir guardando a ordem e ai na segunda vez que executar o programa sabe quais páginas vai usar, mas isso não funciona porque as páginas que um processo usa não é necessário o mesmo, porque them ifs e tals, depende de valor de variável> if executam trechos de códigos diferentes

Algoritmo do uso mais antigo/menos recente/menos recentemente usada/ lru – least recente used – Página 14

Vítima é o uso mais antigo no passado

Olha o uso das páginas no passado

Embora o LRU seja implementável, na prática ele não é porque falta informação para ele rodar. Ele precisa saber a sequência exata de páginas usadas pelo processo, mas isso não é conhecido o SO não tem essa informação. SO pode saber de coisas quando ele roda, mas se ele não rodar ele não sabe. As páginas que não tiveram interrupção o SO não sabe delas. O hw sabe porque é ele que acessa, mas ele não guarda essa informação (páginas que ele aceitou em sequência de tempo)

Teoricamente implementável, mas na prática não rola

Na vida real existe uma simplificação do LRU, sem resultados tão bons, mas melhores que o FIFO

Tipo de algoritmo

Not recentlu used (não recentemente usado) NRU -VIDA REAL – Página 14 verso

Simplificar em não escolher a mais antiga (LRU) e sim uma só antiga. Divide a sequencia de acesso em 2 partes, acesso recente e antigo. Qualquer uma com acesso antigo pode ser vitima, mas não necessariamente é a mais antiga

Se usa o bit acessado(usado) na tabela de páginas, ele lembra o bit sujo, ele é ligado quando a página sofre algum tipo de acesso e o hw liga automaticamente (mesma coisa do outro, liga na TLB e vai pra memoria quando alinha é descartada). Com esse bit o SO consegue saber se a página sofreu acesso, mas ele não fala se é recente ou não. De tempos em tempos o SO zera o bit acessado das páginas, se ele tiver ligado significa que ele foi usado recentemente porque ele já apagou tudo e isso é considerado recente.

Windows – Algoritmo da segunda chance – Página 14 verso

Evolução do algoritmo da fila, ele mantém uma fila e verifica o bit acessado do primeiro da fila, se o bit acessado tiver o valor 0 ela é vítima de fato, mas se tiver o valor 1 o SO entende que essa página teve um acesso recente e então ele não faz essa página ser vítima (da a segunda chance) e coloca a página no fim da fila zerando o bit acessado

Unix Original- Algoritmo do relógio Página 15

O critério é a ordem numérica das páginas, a primeira que ele verifica é a 0, se o bit estiver 1 ela não vai ser vítima e vai zerar e vai avançar para a seguinte e vau fazendo achar uma com 0. Tem um ponteiro que aposta para as páginas e assim que ele verifica.

Desliga é o SO, quem liga é o hw < Bit

Quando o hw não tem o bit acessado – ARMY

Como vai para fazer NRU? Se usa um macete, o SO tem uma tabela auxiliar e ela tem o bit acessado, ai quando tem a segunda chance o SO zera esse bit, mas nada adianta porque o hw não conhece essa tabela. Para contornar isso o SO mente o bit válido e zera, mas na tabela auxiliar ele armazena que ele é válido em outro campo (realmente válido). Ai na interrupção o SO verifica se ela realmente existe, aí ele liga o bit alterado da tabela auxiliar e recoloca o valor verdadeiro na tabela principal no bit válido e a instrução é reiniciada. < Isso na segunda chance é quando acontece o zerar o bit valido > Não é tão bom como LRU, mas é melhor que o da fila

Como o algoritmo executa tendo em vista que tem vários processos?

Alocação de Páginas físicas a processos

(UNIX) Alocação global – Página 15

Não há regra restringindo quantas páginas físicas cada processo deve ter. O algoritmo executa livremente, ele escolhe a página mais antiga de todos os processos se for LRU. Os algoritmos de páginas vítima é o único critério para saber se a página vai ficar na memória física ou não.

Consequência: A vítima pode ser de outro processo do que gerou a interrupção, logo o processo fica com menos uma página e o da interrupção fica com mais uma. É como se tivesse roubado uma página física do processo. A princípio não é ruim, mas digamos quando temos um processo que acessa várias páginas, e ele gera várias interrupções, digamos que outro processo tenha sido vítima, então pode rolar um caso extremo de um processo roubar todas as paginas do outro. Quando esse processo que foi roubado voltar ele vai gerar várias interrupções e vai precisar pegar tudo do disco, aí o local vem com restrição para impedir que um processo fique sem página física nenhuma

Alocação local – Página 15

Existem regra que limita a quantidade mínima de páginas físicas que um processo deve ter. Se um processo tiver com a quantidade mínima e outro gerar interrupção não vai pegar dele, vai ter que pegar do próprio processo ou de outro. Caso todos estejam com a quantidade mínima, então se um processo gerar interrupção a vítima vai ser do próprio.

Dificuldade: Implementação, como definir a quantidade de cada um? Um processo pode ter várias ´páginas lógicas, mas não usar todas, não precisa de muita memória física.

Conceito – Página 15 verso

Working set: Conjunto de páginas lógicas que um processo acessou durante um certo intervalo de tempo

Princípio da localidade: Um processo tende a continuar a usar as páginas lógicas que já está usando. Isso está relacionado com a estrutura de programas que a gente tem, onde estão as variáveis, pilhas etc.> Não é matemática e sim observação

Como juntar esses 2 conceitos?

Working sets tendem a ser estáveis.

Quantas páginas no mínimo um processo deve ter? Quantidade mínima de páginas físicas de um processo é igual tamanho do working set do processo.

Se for menos não cabem as páginas lógicas e vai ficar precisando mais

Se for igual está lindo, enquanto o ws vai estar estável sem gerar interrupção, quando mudar (ir pro instável) vai ter interrupção e via precisar páginas do disco. Mas quando reestabilizar não vai ter mais interrupção.

Quando tem mais páginas que o ws tem poucas interrupções. Mas tem alocado mais memória física do que o processo de fato precisa, ou seja, desperdício de memória física. Mas se alocar um novo processo ele não vai encontrar páginas físicas, pois tem processos com reserva exagerada e ai da erro pra criação do processo novo. Ai o SO toma uma ação, quando o SO não consegue reservar mais páginas físicas a algum processo ele supõem que todos os processos estão com reserva exagerada. Então o SO diminui a quantidade de páginas físicas de todos os processos. Isso pode gerar problema nos que estavam com a quantidade exata eles passam pra primeira situação e aí o SO retorna pra quantidade que estava antes da interrupção << Windows

Esse problema não é tão grave hoje porque as memórias estão maiores

Unix antigamente tinha tipo essa trava manualmente

Problema: Como obter a sequência? E aproximação, não tem definição, vai variando até ter pouca interrupção

(Página 16) Thrashing - Arrastamento > Competição intensa pela memória (Pouca memória e ai eventualmente cai) > Grande bloqueio pra pegar página em disco e a máquina fica lerda > Ele é muito mais fácil de ocorrer com alocação global, cada processo rouba de outro e ai sai comendo interrupção. Local em tese deveria segurar isso, um pode rodar mal, mas não propaga.

Solução (UNIX): É usar swap de processos. Quando o SO percebia que vários processos estavam gerando várias interrupções, o SO escolhia um ou mais processo para jogar totalmente em disco. Diminui a competição por memória e libera a memória.

Bloqueio de buscar a página em disco, então demora um cadin mais > sem paginação não tem

Não altera o bit sujo em memória porque gasta muito e só ligar na TLB. Quando a linha for descartada ele joga na memória.

A última página de código vai ter que ser parcialmente zerada porque ela pode ter o conteúdo da parte anterior e a última página de código tem fragmentação interna e o restinho vai precisar zerar

Mais azarado da tabela invertida é melhor do que os 4 acessos

Segmentação usando o disco Segmentação sob demanda – Página 16

Supomos que precisamos alocar um segmento novo. A logica é escolher um segmento que está na memória e joga para o disco. Ai o bit válido vira 0 e ai libera memória. Ai pode alocar outro segmento, a tabela de segmento cria outro campo.

Tem um processo invalido agora, se tentarem executar ele vai gerar interrupção e o so vai tratar ela e carregar o seguimento de volta e se não caber precisa de outro vítima.

Parecido com paginação, var precisa salvar no disco, código não porque tá no executável. Variável talvez precise, tem que ver.

Diferença da paginação

É segmento e não página, ou seja, segmentos não tem o mesmo tamanho e são tipicamente maiores. Então para trazer de volta um segmento para a memória precisa escolher um ou mais segmentos pra sair, na paginação não tem isso, qualquer página serve porque todas tem o mesmo tamanho. Além disso segmento demora mais para ir para disco porque são maiores que página, isso pode diminuir aumentando a quantidade de segmentos, em vez de uma única área de código se quebra ela, além de ajudar no tamanho e se não quebrar quando for executar necessariamente vai precisar trazer ele de volta do disco.

Usado quando não tinha paginação

TLB – registrador de segmento (não é transparente pra quem usa como a TLB (transparente pro processo) > programa em linguagem de máquina? Desempenho é o mesmo papel)

Troca de segmento swap de segmento, escolha de segmento vítima idealmente é algo tipo NRU, ir pro disco e memoria, não precisar salvar código em disco, se tiverem vários segmentos não precisa carregar tudo de uma vez, só o primeiro de código, os outros são por demanda. Pode por todos ás áreas de código combir valido igual a zero e depois no primeiro acesso criar > o que tem em comum

Compactar a memória pra trazer um segmento gg >paginação não tem isso

----------**P2**----------

Entrada e Saída

É quando a CPU envia (saída)/recebe (entrada) um conteúdo para/de um dispositivo, que não é a memória. Exemplo: Placa de vídeo, mouse, impressora etc.

Como é feita? – Página 1

Depende do computador. A **Intel** usa através de instruções específicas de entrada de saída.

(**Recebe dado**) IN Registrador, Identificador_de_Dispositivo > Recebe dado de um dispositivo e coloca em um registrador.

(**Gera um dado pra fora**) OUT Identificador_de_Dispositivo, Registrador > Dado vem do registrador e é enviado para um dispositivo.

Identificador_de_Dispositivo também pode se chamar de número de porta, sendo que cada dispositivo tem dois ou mais números para fazer a comunicação com o mesmo. Além disso dependendo do dispositivo há uma padronização da quantidade. > Porta de dados e para verificar status é na porta de status

Entrada e Saída mapeada na memória – Página 1 verso

Existe um intervalo de endereços de **memória física,** que não possuem memória, reservado para fazer comunicação com os dispositivos, sendo **o hardware que faz esse controle**. Supomos que esse intervalo seja de 1000 à 1999.

MOV [1000], AH | OUT 3F8H, AH (Intel)

Em tese estaria alterando o endereço 1000 de memória, mas como ele faz parte da faixa reservada para entrada e saída ele não tem memória. Então ele seria como um dispositivo e estaríamos enviando dados para ele. Se equivale a um OUT (estar enviando dados para um dispositivo), mas em vez do destino ser um número da porta, é um endereço de memória sem memória.

Por que se usa assim em vez das instruções IN e OUT?

Isso é usado tipicamente quando a CPU não tem as instruções de entrada e saída, então se usa assim ou não dá para fazer a comunicação de E/S.

Exceção da Intel não usar as instruções de IN e OUT e sim mapeamento na memória

Em caso da placa de vídeo não se usa. A placa tem dois módulos, trabalho e texto, sendo que esse só tem texto e é o qual a máquina boota. Depois de fazer o boot ele entra no modo gráfico, onde se tem os pixels na tela e podemos mudar a cor de cada um (matriz de pontos). Para mudar cada cor do pixel precisamos informar as coordenadas dele e a cor nova, logo, 3 dados diferentes que precisamos mandar para a placa de vídeo para fazer a alteração. Ou seja, seriam necessários 3 OUT, uma para x, outro para o y e outro para a nova cor, para cada alteração da cor de um único pixel. Esse processo demora e se precisarmos alterar vários pixels vai demorar mais ainda.

Para não gastar tanto tempo, se mapeia em memória a placa de vídeo. Existe um intervalo de memória que simboliza a placa de vídeo em modo gráfico, então a cada 3 bytes da memória mapeada simboliza a coordenada (x e y) de um pixel. Ou seja, para o endereço de memória usado está sendo indicado o pixel que terá a cor alterada e só precisa informar a cor, logo, uma solução mais eficiente.

Tipos de informação que circulam nas instruções de entrada e saída – Página 1 verso

Existem 3 tipos:

- Dados (Entrada/Saída – Varia conforme o dispositivo): Informação relevante para alguém. Exemplo: cor do pixel, o que vai ser impresso na impressora, qual o lugar que o mouse foi quando você mexeu nele;
- Controle (Saída): São as ordens dadas pela CPU para um dispositivo, que muda o comportamento dele. Exemplo: Para se mudar a resolução da placa de vídeo se usa uma ordem, mexer a cabeça magnética do disco magnético é uma ordem;
- Status (Entrada): É o a situação que o dispositivo está em determinado momento que é enviada para a CPU pelo próprio dispositivo, ou seja, a CPU recebe o status do dispositivo. Exemplo: a impressora ter papel é o status que ela retorna quando o SO pergunta.

Como o controle e status estão sempre em ordem trocadas, eles compartilham o mesmo identificador. Ou seja, se faz saída é controle e se faz entrada é o status.

No caso da Intel todo o dispositivo tem pelo menos 2 portas, a de dados, que se recebe ou envia dados para o dispositivo, e a de controle/status, que se sabe o que é feito pelo tipo da instrução (IN ou OUT).

Sincronismo CPU x Dispositivo – Página 2 verso

Os dispositivos podem ter vários status, sendo um dos mais importantes a possibilidade dele se comunicar com a CPU. A verificação da capacidade de se comunicar tem dois lados, um de entrada e outro de saída. Quando é entrada significa que o dispositivo tem um dado novo, por exemplo, alguém digitou uma tecla no teclado. Já no caso de saída quer dizer que o dispositivo é capaz de receber um dado novo.

A CPU é capaz de enviar ou receber dados mais rapidamente do que o dispositivo consegue processar (receber ou enviar). Sendo assim é necessário haver a **verificação do status**, onde o SO testa o status do dispositivo antes de enviar para saber se ele pode realmente fazer a E/S.

Esse mecanismo tem duas variações, a **espera ocupada** e o **Polling**. O primeiro consiste em o código do SO ficar em loop até que o status do dispositivo mude/consiga se comunicar, o que gera desperdício de CPU. Já o segundo verifica o status do dispositivo de vez em quando, por exemplo a cada segundo, e quando o status muda faz o envio. Sua desvantagem é perder um tempo em que o dispositivo já trocou o status e o SO ainda não verificou.

O que é realmente usado é **sincronismo por interrupção**. O SO manda o disco ler um conteúdo, o disco lê, porém ele irá demorar na leitura, sendo assim o processo que pediu a leitura fica bloqueado. Os dados lidos pelo disco irão para sua memória interna e no fim da leitura ele gera uma interrupção. O SO trata essa interrupção e traz os dados da memória interna do disco para a memória principal do computador, desbloqueia o processo que está esperando e acaba a interrupção. Agora o processo desbloqueado poderá usar os dados. > Quem gera a interrupção é o dispositivo

Dispositivos comuns que utilizam esse mecanismo são mouse, teclado, impressora. Porém o disco não usa, somente a “variação”, pois ele lê muitos dados. As instruções de E/S não leem muitos bytes (4 usualmente), sendo assim, é necessário repetir essas instruções várias vezes, ou seja, fazer **um Loop no IN e MOV** (em cada execução se avança em 4 o endereço dele).

Essa solução irá funcionar, mas a CPU não é muito rápida para fazer cópia de dados, pois ela é muito genérica (faz de tudo). Logo, é mais fácil criar um componente que só faz cópia de dados e é muito mais rápido.

OBS: **Rotina tratadora de interrupção**: Se lê da porta de dados e não precisa checar o status, visto que a interrupção só ocorre quando muda o status. No entanto, o registrador que sofre a interrupção terá o valor alterado, então a rotina que trata interrupção precisa preservar o valor original dos registradores e restaurá-los antes da rotina acabar. Isso é normalmente feito colocando na pilha o valor dos registradores.

PUSH BH
IN BH, 3f9H; Porta de dados

MOV [2000], BH

CALL DESBLOQUEIA_PROCESSO >Também precisa preservar os valores

POP BH

IRET; //Acabou o tratamento

O sincronismo por interrupção é bom, mas para maiores quantidades é usado o **sincronismo por DMA**. Onde se usa o controlador de DMA (Direct Access Memory) que coloca o dado na memória, sem ajuda da CPU. Ele se liga aos dois barramentos (E/S e memória) e transfere os dados dentro da memória.

Essa solução tem uma eficiência de tempo, pois o dado estará mais rápido na memória do que se fosse por interrupção, onde existiria o loop. A única consequência é que o conjunto de fios do barramento é o mesmo, então só se pode ter um único uso, ou seja, se o controlador de DMA estiver pondo dados na memória, a CPU não vai conseguir usá-la ao mesmo tempo. Logo, a CPU não consegue usar a memória o tempo todo.

Além disso, o controlador de DMA é mais complexo. Quando se é feito uma E/S de disco por DMA, a ordem do disco é parecida, o SO manda o disco ler X lugar. Porém o SO também precisa avisar o controlador de DMA que o disco vai fazer uma leitura, que o conteúdo lido tem que ficar na posição Y e quantos bytes vão vir do disco. Quando o disco termina de ler, ele avisa ao controlador que acabou e o controlador faz a transferência 4 bytes por vez para a memória. Ou seja, a preparação do DMA é maior, precisa também ter ordem para o controlador, não basta somente a do disco. > Transparente para a CPU e é o Hardware que controla

Importante lembrar que tem um processo esperando os dados para voltar a rodar, logo, tem que haver um desbloqueio dele. Então o **controlador de DMA** gera uma interrupção quando ele acaba a transferência de dados, a CPU é interrompida e executa uma rotina que trata essa interrupção e desbloquear o processo. >> polling e controle de DMA precisa preparar o disco

Disco Magnético – Página 3 verso

Unidade de gravação é o mecanismo que consegue saber se o conteúdo que está sendo lido de um setor corresponde ao mesmo valor que foi salvo nele. Pois, como é analógico, é possível que possa ter ocorrido algum problema de hardware e tenha se perdido alguma informação.

No final de cada setor existe a **informação de redundância**, que o local onde é salvo alguma conta feita em cima dos bytes que precedem ele. Quando se lê o setor, é lido os bytes e em seguida esse campo, o hardware soma os bytes de dados (refaz a soma), se os bytes não tiverem mudado o valor deve ser igual, logo, o hardware entende que não houve perda de conteúdo.

Tempo para a leitura de um setor – Página 4

Primeiro é necessário que a cabeça magnética se mova até alcançar a trilha do setor, mas o disco não para de rodar, então o setor já se locomoveu. Em seguida é necessário esperar o setor passar embaixo da cabeça e pôr fim a cabeça irá passar em cima do setor (ler). O primeiro passo se chama **Tempo de Seek**, tempo para a cabeça magnética chegar na trilha onde está o setor que se quer acessar. O segundo se chama **Tempo de Latência**, tempo para a cabeça começar a passar sobre o setor que se quer acessar. O último se chama **Tempo de Transferência**, tempo que demora para a cabeça passar ao longo do setor (conteúdo é copiado da sequência magnética e é posto na memória interna do disco). A soma desses tempos se chama **Tempo de Acesso**.

Tempo médio de latência - Página 3 verso

No **melhor caso** é quando a cabeça magnética chega quando o setor começa, ou seja, **não se gastou latência (latência = 0)**. Já o **pior caso** a cabeça chegou no setor, mas perdeu o início do setor, então é necessário esperar o disco dar uma volta para pegá-lo no início. Logo, o **tempo de latência é uma volta**.

Latência média = 0 + Tempo de 1 volta/2 = Tempo de ½ volta

Tempo de transferência – Quantidade de setores por trilha

Sequenciamento – Página 4

Quando se lê setores separados temos 3 tempos de Seek e 3 de latência, mas quando eles estão em sequência só tem Seek e latência para ler o primeiro e os outros não possuem eles.

Se coloca o setor 9 no verso do 1, pois é só usar a cabeça de baixo e não gasta tempo de seek ou latência. Já o 17 volta para cima, embaixo do 2 (se não tiver outro prato), se gasta um seek e uma latência e tem um tempo viável para chegar nele.

Visão linear do disco (Visão Lógica) – Página 4 verso

Ocorreu uma simplificação e agora o disco é visto como um vetor de setores pelo SO. Quando se quer ler um setor é só dar o número de índice.

Com a visão linear, quando escolhermos setores em sequência eles correspondem a setores interessantes para pôr um arquivo na visão física. Os setores do 1 ao 16 estão na mesma trilha e não se precisa fazer seek, o que facilita para o SO, pois ele está preocupado em alocar arquivos em sequência, visando em deixá-los na mesma trilha (sem mexer a cabeça).

Conjunto de trilhas que tem o mesmo raio – Página 4 verso

Esse conjunto de trilha se chama cilindro. Esse conceito é importante porque quaisquer dois setores localizados no mesmo cilindro conseguem ser lidos sem a necessidade de seek. > Melhor do mundo para um arquivo, pois não precisará fazer seek intermediários entre os setores do arquivo.

Tamanho dos setores – Página 5

Quantidade de bytes em um setor é o mesmo, mesmo que tenha tamanho diferente. Isso não é bom porque uma área maior pode guardar mais magnetismo (mais bytes), então antigamente existia um desperdício de capacidade magnética nos setores maiores. > Demora o mesmo tempo para ler os dois (Conversor analógico digital trabalhava com a mesma taxa de conversão)

Atualmente os conversores analógicos digitais trabalham taxas variadas, então pode-se ler setores com taxas diferentes. Então os discos hoje em dia, nas trilhas mais externas possuem maior quantidade de setores, logo, não tem mais o mesmo ângulo os setores, mas os setores possuem aproximadamente o comprimento/possuem mesma área, sendo assim não tem desperdício de capacidade magnética. O problema que o tempo de leitura de cada setor não é mais constante, então para ler os setores das trilhas mais longes do centro é gasto menos tempo (são mais rápidos).

Arquivo – Página 5

Tem mais alterações em arquivo porque é mais distante da influência do Hardware, depende mais do SO.

Arquivo é algo que possui um nome, conteúdo (dado que mandou salvar) e informações de controle (por exemplo data da última alteração, onde o arquivo está no disco). Onde ele está no disco não é importante termos de funcionalidade e sim em termos de desempenho.

Nome

Os arquivos têm várias características que podem variar conforme o Sistema Operacional, neles temos**: tamanho máximo**, **maiúscula e minúscula**, tem SOs que só usam maiúscula (**MSDOS**, **MainFrame IBM – Z/OS**), os que mesclam diferenciando os casos (**UNIX**) e os que não diferenciam (**Windows**).

Conteúdo do arquivo

O conteúdo do arquivo tem grande variação de acordo com o SO. Sempre existiu arquivos texto, mas antigamente eles também eram usados para base de dados, visto que não tínhamos Oracle e era necessário fazer programas para processamento de dados, então eram usados os arquivos, onde o conteúdo é uma **sequência de registros** (**Z/OS**). Na hora que ele cria o arquivo ele define como é o registro, com tamanho fixo ou variado ou equivalente ao que é uma tabela hoje. Caso fosse uma "tabela" como se achava um registro? Pode-se ir de um em um, mas essa solução demora. Atualmente temos a indexação, onde o conteúdo é indexado e existe uma estrutura auxiliar, que guarda o index, e quando se procura o registro usando o campo indexado é muito mais rápido., porque o index indica onde está o registro. Como isso não era comum de se fazer, quem bolava isso trazia para o SO e ele que indexava. Então não só se criava o arquivo e definia o que era o registro, também podia definir campos dentro dele para os quais seriam criados index.

Como consequência disso o SO fica mais poderoso, mas por outro lado pode-se trazer uma deficiência para ele, pois ele fica mais complexo para lidar com arquivos com index. Depois isso se inverteu e o arquivos ficaram mais simples e programas que lidariam com a complexidade e não o SO, deixando-o, SO, mais eficiente. Nessa visão os arquivos viram (para o SO) **sequências de bytes** (**UNIX/Windows 98 (antigo)**), onde ele é o mais simples que se pode existir e qualquer interpretação sobre o formato dele é feito pelo programa. > Simplifica o SO e a complexidade vai para os programas, isso dá uma aliviada com a biblioteca compartilhada, sendo assim não precisa entender tudo. > SO não conhece nada sobre o formato dos arquivos, um arquivo jpeg é equivalente para o núcleo do SO a um txt e que é equivalente a todos os outros. Isso pode-se ver quando a gente renomeia um arquivo jpeg para txt e funciona

Também existe o conceito usado pelo Windows (atual), que seria uma evolução do conceito anterior, chamado de **várias sequências de bytes**. Isso seria equivalente a estar lendo arquivos de diferentes conteúdos, mas independentes entre si, pode-se editar um sem alterar o outro, tornando mais flexível o conceito de conteúdo do arquivo e permitindo que tenham vários conteúdos simultâneos independentes. Sua aplicação original era em arquivo executável, onde se tem uma parte código e outra recursos (Strings que são as mensagens, labels e etc.), que podem variar de acordo, por exemplo, com a língua. A ideia da separação é permitir separar o que é código do que é recurso, permitindo a edição das Strings sem precisar mexer no código executável. Podendo até haver permissões diferentes por pessoa, por exemplo um tradutor só teria acesso para editar os recursos que trocam de língua. Atualmente essa possibilidade é mais usado para pôr **informações de controle extra que não são fixas**, por exemplo, você baixa um arquivo executável usando um browser e ele marca que esse arquivo é baixado da internet. Quando se manda executar no explore pode mostrar um alerta de que o arquivo foi baixado da internet e se quer seguir com a execução.

Diretório – Página 5 verso

O propósito dos diretórios/pastas é organizar arquivos. E possuem uma coisa em comum, pode-se crescer ou diminuir o tamanho de um arquivo conforme o tempo e o diretório também, basta colocar ou tirar arquivos. Para não implementar essa logística várias vezes, o SO trata o diretório como se fosse um tipo especial de arquivo. Além disso o diretório possui as outras características do arquivo, variando o conteúdo.

Conteúdo do diretório

Há três opções para o que o diretório tem de conteúdo. A primeira é para cada arquivo no diretório, ele **guarda o nome do arquivo e uma referência a estrutura de controle desse arquivo.** Uma parte do disco é ocupada pelo vetor de estruturas de controle, onde em cada setor pode conter várias estruturas de controle, portanto, cada estrutura tem um número, que é a posição do vetor, logo, a referência é esse número. > Cada parte do segmento contém as informações de controle de um arquivo e o diretório só guarda o index, isso ocorre no Windows Novo e no Unix.

No caso do Unix a estrutura de controle de cada registro tem um nome, INODE. No caso do Windows o nome do vetor de estrutura de controle se chama Master File Table (MFT).

A segunda opção é para cada arquivo no diretório ele guarda o **nome do arquivo e as informações de controle do arquivo**, isso é usado no MSDOS e no Windows antigo. As informações de controle do arquivo apontam/referenciam para os blocos do disco que estão o conteúdo do arquivo. Nesse caso o diretório é mais complexo e tem mais dado.

A terceira opção é para cada arquivo no diretório o SO guarda **o nome do arquivo, informações de controle e o conteúdo do arquivo**, que é usado no Z/OS MainFrame e lá é chamado de arquivo particionado (lembra o zip, mas sem comprimir). Nesse caso o diretório contém o arquivo.

Primeira e Segunda X Terceira

Quando ocorre a movimentação de um diretório para outro lugar nas duas primeiras o impacto é pequeno, pois o que o diretório guarda de cada arquivo é bem pouco (pouco bytes). Não se move o conteúdo do arquivo. Já no terceiro tem o conteúdo do arquivo, então tem que mover ele e ainda apagar de onde estava, demora mais do que os outros.

Primeira X Segunda – Página 6

Quando se move um arquivo se copia e depois se apaga o original. Mas suponhamos que não se remova o original, vai parecer que o arquivo está em dois diretórios diferentes e não tem problema no primeiro, pois estaria só duplicando o nome, os dados de controle estão na posição 7.

Isso não é possível na abordagem 2, pois nela é guardado os dados de controle no diretório, se mantivesse estaria duplicando os dados de controle e isso gera problema, porque se usar um arquivo de um diretório e aumentar o tamanho isso não vai ser refletido no outro diretório, pois são independentes.> Na primeira a alteração ia ficar no vetor, então tá ok.

Link (Hard Link) x Cópia

Link é uma referência ao INODE

Cópia, se editar o original não vai se refletir na cópia. Já no Link, se editar o arquivo em um diretório quando entrar por outro diretório para ver o mesmo arquivo ele vai estar editado, pois é o mesmo arquivo nos dois diretórios. > Link não pode no segundo devido as estruturas de controle.

Link gera complicações, quando se apaga o arquivo ele apaga o link entre o diretório e o arquivo. No outro diretório que também tem o arquivo ele vai se manter. Como o SO sabe que não pode apagar o arquivo e sim o link? Ele sabe, pois tem na informação de controle o contador de link, que não informa o diretório e a quantidade de link. Quando é adicionado outro link esse campo é incrementado, quando se apaga ele decrementa. Se solicitar apagar o arquivo que está com esse campo zerado ele irá de fato apagar o arquivo.

Link para diretório – Página 6 verso

Não se pode fazer ciclo/link para cima gerando infinitos caminhos. Isso não é bom, pois se tive rum programa que leia árvore ele irá entrar em loop. Para resolver esse problema o Windows não deixa ter link para diretório, já o Unix deixa o link para diretório se quem estiver fazendo for administrador do computador, pois supostamente o administrador não irá fazer burrada.

Disco Físico/Magnético – Página 6 verso

Partições são pedaços do disco físico e o usuário as vê como um disco diferente.

Volume é aquilo que o usuário vê como um disco.

Volume x Partição

Digamos que tenhamos dois discos físicos que não foram quebrados em partições e que tenham o mesmo nome (terceiro exemplo), teremos 2 partições e 1 volume. Isso é útil porque é como se o volume C crescesse, então você pode mover um arquivo de uma partição para outra sem problemas, mas se ela tivesse outro nome daria problemas, por exemplo, mover um arquivo executável.

Nem todas as versões de Windows permitem romperem a proporção de 1 para 1 entre partição e volume. No Linux não tem na default e é preciso por um software a mais (gerenciador de volume).

Ops: Windows não consegue ver os volumes do Linux, mas o Linux consegue

Definição de bloco – Página 9

Cada volume tem um grupamento de setores em bloco, sendo esses com tamanhos iguais. Antes os setores eram muito pequenos, então teriam muitos, então agrupando eles em blocos têm menos unidades para o SO controlar, logo, é mais simples. O problema é que aumenta a **fragmentação interna**, pois o tamanho do arquivo não é necessariamente um múltiplo exato do tamanho do bloco, se não for no final do último pedaço vai ocorrer uma alocação parcial. > **Similar a paginação**

Como um usuário utiliza arquivos em diferentes volumes? – Página 7

No Windows o usuário utiliza o nome do volume explicitamente. Por exemplo, se ele quiser usar o volume ele faz “D:/blalabla” ou se usa o conceito de volume corrente/atual, onde se quiser usar um arquivo dele não precisa informar o volume.

Também há a forma do usuário não utilizar o nome do volume explicitamente em que ele está, pois de alguma forma o SO sabe em qual volume o arquivo está. O Z/Os Mainframe utiliza o catálogo, quando se cria um arquivo precisa se falar em qual volume ele estará e se cataloga essa informação, o problema é que não podemos ter o mesmo nome de arquivo em dois volumes diferentes. No Unix se usa a montagem, onde se incorpora a árvore de diretórios do volume que se quer montar na árvore de diretório totalmente visível.

Quando a máquina boota ela só tem um único volume visível, o raiz. Para usar qualquer outro precisa montar e precisa do comando mount, onde define um diretório onde você quer que ocorra a montagem e diz o nome do volume. Depois da montagem feita o usuário vai ver uma árvore que é a união das duas árvores e vai até o arquivo navegando.

Comparando a solução do Unix com o Windows, o Windows menciona o volume ao qual o volume está, já o Unix não. Na hora que o SO menciona o volume em qual o arquivo está, há uma maior dificuldade de mover um arquivo de um volume para outro, pois se alguém estiver guardando o nome completo do arquivo e ele se mover, o nome guardado não vai servir mais para achá-lo. Já no Unix é possível, por mais que não trivial, mover um arquivo de um volume para outro mantendo o nome completo (tem que ser o adm). Esse problema é resolvido com a possibilidade de poder ter um único volume em partições diferentes. > Windows já deixa montar

Usuário do Unix não sabe exatamente onde o arquivo está, só se ele fizer a montagem, exceto em caso de linkagem, pois se eles estiverem em volumes diferentes vai dar erro, pois a estrutura de arquivo não comporta isso. A estrutura é composta por nome do arquivo e número de INODE na tabela INODES do volume que

está o diretório, não tem campo que permita apontar para o INODE de outro volume. Esse motivou a criação de um segundo tipo de link tanto no Windows como no Unix (link Simbólico) > O link normal se chama Hard Link

Ops: diretório atual é o que se faz o cd na linha de comando. Para qualquer arquivo que está no diretório atual não precisa dar o nome completo, somente o nome final do arquivo que vai ser possível utilizá-lo. Quando ele não está é necessário usar o caminho a partir da raiz ou fazer o caminho dele.

Link Simbólico – Página 7 verso

Ln -s (link simbólico, o normal é só ln). Existe um arquivo .txt dentro do diretório que guarda o caminho do qual o link simbólico está apontando, que tem uma flag de link simbólico ligada. Caso tente usar/ler esse arquivo em vez de mostrar o conteúdo dele, ele lê o conteúdo cujo caminho está escrito no link simbólico. < Link simbólico referente ao caminho, hard link referência ao INODE

Caso se mande apagar o arquivo que o link simbólico está fazendo referência, ele irá apagar e decrementar o contador de link e se for 0 vai apagar o arquivo. Mas ainda existe o link simbólico que aponta para um arquivo que foi apagado, caso tente usar vai dar erro de arquivo não encontrado.

Atalho x Link Simbólico

Atalho é uma funcionalidade do Windows e link simbólico tem no Windows e no Unix. A implementação do atalho é muito similar ao link simbólico; O Atalho é um arquivo cujo conteúdo é o caminho do arquivo que está sendo referenciado. A diferença é quem implementa, o link simbólico é o núcleo do SO, o que faz funcionar sempre. Já o atalho quem implementa é a interface gráfica com o usuário, também tem atalho nas telas de escolhas de arquivo (DLL), ou seja, são programas que interpretam o atalho, logo, podem não fazer isso bem, então você pode receber o conteúdo do atalho e não do arquivo que ele aponta. Se tentar abrir pela linha de comando irá abrir o notepad com o caminho escrito, pois quem implementa o atalho é o explore.exe e na linha de comando não tem ele.

Ops: Windows tem 2 tipos de atalho para diretório no Windows, o jumpshion (link simbólico para diretório) e o link simbólico para diretório.

Chamadas ao SO para utilizar arquivos – Página 7 verso

São chamadas que permitem ler partes do arquivo para a memória, podendo até fazer ele todo. Para ler o pedaço é preciso identificar o nome do arquivo, qual o pedaço que se quer ler (posição inicial – número byte inicial e a quantidade de bytes), sendo essas informações a origem do dado a ser lido, e o endereço da variável (onde na memória vai ser colocado) – destino do dado a ser lido.

Suponhamos que tenhamos um player de música que vai ler o arquivo por partes

```
Char buf [1000]
Ler (“A.vid”, 0, 10000, &buf);
//mostra na tela o início do vídeo
Ler (“A.vid”, 10000, 10000, &buf);
//mostra na tela o trecho do vídeo
Ler (“A.vid”, 20000, 10000, &buf);
//mostra na tela o trecho do vídeo
Ler (“A.vid”, 30000, 10000, &buf);
//mostra na tela o trecho do vídeo
```

O problema dessa chamada, como se passa o nome como parâmetro, o SO precisa checar se o arquivo existe e as permissões do usuário que pediu para ler. Isso é ineficiente quando o arquivo é o mesmo. Então na realidade existe o conceito de abertura de arquivo, que evita testar toda hora se o arquivo existe.

Int fd; //faile descriptor

Fd = open (“A.vid”,modo_de_abertura); -> retorna um número inteiro

Read (fd, ..., &buf);

//mostra na tela o início do vídeo

Read (fd, ..., &buf);

//mostra na tela o trecho do vídeo

Read (fd, ..., &buf);

//mostra na tela o trecho do vídeo

Read (fd, ..., &buf);

//mostra na tela o trecho do vídeo

Open irá verificar se o arquivo existe uma punica vez. As chamadas seguintes não precisam checar, pois o open já checou e o SO impede que um arquivo aberto seja apagado. > Open lê dados de controle e faz verificações e não conteúdo, o conteúdo é a chamada read

No Windows há a chamada ao SO chamada open e existe a opção de menu file (faz um open e um read) open que são diferentes. O segundo lê da memória o conteúdo do arquivo, já a primeira não.

Posição corrente do arquivo

Leitura em sequência – Read em sequência

Int fd; //faile descriptor

Char a [10000]

Fd = open (“A.vid”,o_rdonly); -> somente leitura

Read (fd, a, 10000); -> Vai ler os primeiros 10k bytes para a variável a e o SO avança a posição atual para o byte seguinte ao último byte lido/primeiro byte não lido

//faz algo com o array

Read (fd, a, 10000); -> Ler mais 10k bytes referentes a posição atual (byte seguinte ao último lido)

...

Close (fd)

Posição corrente sempre aponta para o primeiro byte que você não leu. Ele sempre lê coisa nova e não precisa se preocupar com a posição.

Escrita em sequência

Int fd; //faile descriptor

Char a [1000]

Fd = open (“A.dad”,o_wdonly); -> somente escrita

//atribui algum valor ao array ‘a’

Write (fd, a, 1000); -> Escreve/altera os mil bytes iniciais do arquivo e a posição inicial avança

//atribui algum valor ao array ‘a’

Write (fd, a, 1000); -> Supondo que o arquivo só tenha 1500 bytes, será alterado os 500 finais existente e irá aumentar o arquivo em mais 500, pois foi escrito além do tamanho original dele -> Escrita além do fim do arquivo aumenta o mesmo

...

Close (fd)

Open não lê o conteúdo do arquivo da memória, quem faz isso é o read. Ele lê os dados de controle na memória e o SO marca o arquivo como aberto e não permite que ele seja apagado enquanto ele estiver aberto. Quando fechamos (close) é porque não precisamos mais desses dados de controle e o espaço é liberado. Close também passa a permitir que o arquivo seja apagado.

Open do Windows ele regula o uso simultâneo do mesmo arquivo, quem abre ele pela primeira vez pode permitir ou vetar que outro processo também o abra. Então o close também permite que outro processo use o mesmo arquivo.

O close não é obrigatório, usamos para indicar que não queremos mais usar o arquivo, pois quando o processo morre ele implicitamente faz o close do arquivos que abriu.

Chamada que muda a posição corrente – Acesso não sequencial / Acesso Aleatório/ Acesso Randômico

Fd = open (“A.dad”,o_rdonly); -> somente leitura

Lseek(fd, 5000(nova posição), Seek_begin(onde começa)) -> Muda a posição atual para 5000. Não faz o seek / Seek_begin é a partir de onde conta a nova posição

Read (fd, a, 100); -> Lê do arquivo para variável a 100bytes a partir de 5000 e avança a posição atual para o primeiro byte não lido

Lseek(fd, 2000, Seek_begin)

Read (fd, a, 100);

Isso seria para por exemplo ler uma lista de empregados, mas os nomes que eu quero não estão em sequência. Ou seja, se quer ler uma parte do arquivo que não começa no início.

A terceira posição do lseek representa a partir de onde conta a nova posição e pode ter 3 valores, do início do arquivo, da posição atual (pode ser positivo-anda pra frente- ou negativo -anda pra trás-) ou do fim do arquivo.

Um exemplo de quando se usa o fim do arquivo é o arquivo de log, onde sempre se escreve uma informação nova no fim.

Fd = open (“A.log”,o_wronly); -> somente escrita

Lseek(fd, 0, Seek_end)) -> Muda a posição atual para 0 no final do arquivo, ou seja, byte seguinte ao último byte do arquivo

Write (fd, a, 100); -> Cresce o arquivo em 100 byte e a posição atual avança para o fim

Os chamados que estamos vendo são chamados ao SO e são bem simples. Porém temos muitas vezes bibliotecas que internamente usam esses chamados e que dão uma rotina da biblioteca mais simples para se usar para um arquivo específico, ou seja, simplifica o uso, visto que não precisamos lidar diretamente com chamados. Por exemplo, a leitura do arquivo jpeg, que é um formato complicado. Na biblioteca DLL do Windows quando se pede para ler esse arquivo ela lê tudo para memória, faz a descompressão e internamente faz todos os chamados.

Arquivo Esparso – página 8

Suponhamos que temos um arquivo de 10240bytes

Fd = open (“B.dad”,o_wronly); -> somente escrita

Lseek(fd, 2048, Seek_end(onde começa)) -> Muda a posição atual para 2048 após o fim

Write (fd, a, 1024); -> Essa ecrita depende do SO, alguns dão erro e outros permitem. Ou seja, o arquivo vai ter um buraco e depois a escrita. Esse espaço pulado não existe no disco, mas se mandar ler essa parte o SO retorna zeros em binário como se tivesse algo salvo nele

Arquivo esparso, não é um arquivo contínuo, tem um espaço/buraco entre as partes, mas o SO finge essa parte existe e preenche com 0. A sua definição é o

espaço ocupado pelo arquivo no disco é menor que o tamanho lógico do arquivo. (11264 (tamanho do arquivo) < 13312(tamanho do arquivo))

Quando se aloca um arquivo em disco é preciso respeitar a unidade de alocação, os blocos. Supondo que temos um arquivo não espaço que tenha 1500 bytes, serão necessários 2 blocos de 1024 bytes para armazenar, no entanto o segundo não vai ser usado completamente. O espaço ocupado no disco vai ser maior ou igual ao tamanho lógico do arquivo, por conta da fragmentação interna. (2 blocos - 2048 bytes (espaço ocupado no disco) > 1500 bytes (tamanho lógico do arquivo). É igual quando não tem fragmentação interna.

No arquivo esparso é o contrário, é menor pois tem trechos do arquivo que não existem em disco.

Ops: É mais vantajoso o SO alocar todo o arquivo em blocos sequenciais, porque a leitura é mais rápida, mas se não estiverem ainda vai funcionar pois quem tá usando não precisa saber disso, só foca na visão lógica.

Quando o arquivo esparso é bom

Digamos que tenhamos uma matriz (array de duas dimensões) com 1024 linhas, 1024 colunas e cada célula guarda um número inteiro de 4 bytes. O espaço ocupado por ela na memória é de 4 megabytes.

Se ela for uma matriz especial, ou seja, maior parte das células com valor 0, ela pode ser salva de um jeito muito mais esperto do que mandar fazer o write da matriz toda.

```
Fd = creat (“m.dad”, bits_acesso)
For (i=0; i<1024; i++)
    If (linha_só_tem_zeros(m[i])
        Lseek (fd,1024(tamanho da linha)*4(bytes da célula),seek_corrente) -> Pula a linha que tem o valor 0
    Else
        Wtrite(fd,m[i],1024*4)
Close(fd)
```

Esse jeito seria criando o arquivo esparso, pulando as linhas que só tem 0. Caso mande ler a matriz toda vaia aparecer certinho, pois as linhas puladas vão ser lidas como 0.

Volume (Vetor de blocos) - Controle de alocação – Página 8 verso

Alocação Contínua (Mais antigo e mais simples) – Página 8 verso

O SO guarda apernas dois números para cada arquivo, ou seja, apenas dois números inteiros controlam os arquivos alocados. Esses números são bloco inicial e quantidade de blocos, também é salvo o nome do arquivo. Para ele funcionar o arquivo tem que ser contínuo (ocupando blocos em sequência).

Um problema é que se você quiser aumentar o tamanho de um arquivo do início ou meio não vai poder, pois ele vai parar de ser contínuo. Sua vantagem é que ele tem o melhor desempenho possível.

Alocação encadeada (Não é usado) – Página 8 verso

Ela possui uma lista encadeada dos blocos que compõem o arquivo, como se o bloco tivesse um campo a mais no final que é um ponteiro para o próximo bloco. Essa alocação não tem problema ou crescer o bloco, só trocar o ponteiro do bloco, ou seja, permite descontinuidade.

Ela é melhor que a alocação contínua, mas não é tão boa. Tem um problema para acesso não sequencial do arquivo. Para ler um bloco de um arquivo precisar saber onde ele está e por ser uma lista encadeada não é trivial, tem que percorrê-la

toda. Tem que por todos os blocos antecessores na memória e ler cada um até o pegar o campo de controle que diz qual é o próximo.

Alocação encadeada com estrutura de controle separada (similar a anterior) – MSDOS e Windows

Ao invés de colocar o ponteiro para o próximo no final de cada bloco, ela pega um ou mais blocos do volume para serem blocos de controle que irá guardar o ponteiro para o próximo de cada arquivo (File Allocation Table – FAT). > Fat do Windows fica no início do volume

Com esse mecanismo temos uma grande diferença em referência a alocação encadeada. A ideia é que o SO coloque esse bloco de controle na memória fazendo com que pegar um encadeamento seja mais fácil, ainda será necessário percorrer o bloco todo, mas em memória é muito mais rápido.

O problema dessa alocação é com volumes grandes, supomos que tenhamos um volume de 1 TB e blocos de 1kb.

Quantidade de blocos = Tamanho do volume/ Tamanho do bloco = 2^30 (1 Giga blocos)

Espaço ocupado pela FAT = Quantidade de registros * espaço gasto por elemento = 1G * 4B = 4GB

Na Microsoft tem a FAT 12(1byte e meio), FAT 16 (2bytes), FAT 32 (4bytes)

A FAT passa a ser uma estrutura rápida quando é carregada na memória, mas quando ela tem 4 GB ela precisa gastar muita memória para ser carregada. Então quando se quer usar a FAT num volume grande, a Microsoft usa um bloco muito grande, diminuindo assim a quantidade de bloco e a FAT fica menor. No entanto isso pode gerar fragmentação interna.

Ops: Se o volume tiver mil arquivos a FAT vai controlar os mil encadeamentos/alocação dos mil arquivos. Porém todos os arquivos não vão ser usados ao mesmo tempo, pelo contrário, poucos são usados (os abertos), mas a FAT vai por mesmo os sem uso na memória, logo, há um desperdício (guarda controle de alocação de arquivos que não estão sendo usados no momento).

Então tanto quanto o Windows e o UNIX tem uma estrutura de controle separada para cada arquivo e só vai para a memória do arquivo que está aberto, ou seja, se gasta menos espaço de memória.

Alocação Indexada (UNIX) – Página 9 verso

Não possui uma estrutura de alocação única, é como se fosse um índice para cada arquivo. Para cada arquivo tem um array que guarda os números de blocos onde o arquivo está.

A dificuldade é que o array pode mudar muito de tamanho.

Para arquivos pequenos o Unix já tem uma estrutura que guarda os dados de controle de um arquivo (INODE) e em certo lugar no disco há um lugar que guarda o vetor de INODE. Esse inode tem vários campos e um deles é um array que tem 10 elementos, que guarda números de blocos. Se o arquivo é pequeno todos cabem no INODE.

Para arquivos maiores vai ter também o INODE com os mesmos 10 elementos, mas não vão caber todos os blocos. Então se usa um bloco vazio do volume e ele irá ampliar a lista do INODE(a contagem começa no 11, como se fosse a continuação do array do INODE), que é chamado de bloco índex ou de indereção. No INODE existe um bloco que guarda esse index, chamado de bloco de indireção.

O problema é que o bloco de indireção não é infinito. Quantidade de elementos no bloco de indireção = tamanho do bloco / espaço ocupado para cara elemento -> 1KB/4B =256 elementos/registros/números de bloco

Caso o arquivo tenha mil blocos o INODE vai guardar os 10 primeiros blocos, por o bloco de indireção. A diferença é que se terá outro bloco de indireção no volume, chamado de bloco de dupla indireção, que guardará o número de bloco de indireção.

Ou seja terão mais blocos no volume de indireção e um de dupla indireção que referencia eles. Esse o número bloco de dupla indireção também estará no inode junto com o bloco de indireção.

O bloco de dupla indireção consegue guardar 256 blocos de indireção e cada bloco de indireção consegue guardar 256 blocos. Ou seja, a quantidade max de blocos é 10 (número no INODE) + 256 (primeiro bloco de indireção) + 256 * 256 (256 blocos de indireção que são apontados pelo bloco de dupla indireção)

Tamanho máximo do arquivo = quantidade máxima de blocos * tamanho dos blocos -> 64K blocos * 1KB = 64MB. Que é um arquivo grande, mas não gigante

Para arquivos gigantes existem a tripla indireção, que guarda os números de blocos de dupla indireção. Quantidade máxima de blocos = 10 + 256 + 256^2 + 256(bloco de tripla indireção)*256^2 (outros blocos de dupla indireção) = 16M blocos

Tamanho máximo do arquivo = 16GB

Para arquivos maiores é comum se aumentar o tamanho do bloco. Para um bloco de 2KB:

Quantidade de números de bloco em um bloco de controle = 2KB/4B=512 números de bloco

Quantidade máxima de blocos de um arquivo = 10 + 512 + 512^2 + 512 ^3 =128M número de blocos

Tamanho máximo do arquivo = 128M * 2KB = 256GB

Nesse tipo de alocação não importa se o arquivo tiver contínuo ou não, pois se ele tiver mil vai precisar de mil

Alocação por extensão (Windows e Main Frame) – Página 11 verso

Leva em conta o fato de o arquivo ser contínuo ou não, e o nome tem a ver com o fato dela ser uma evolução da alocação contínua. Onde tem uma estrutura que informa o bloco inicial e quantidade de blocos para cada arquivo.

No entanto há uma dificuldade na alocação não sequencial. Na alocação indexada é muito mais simples saber onde está o 28ª bloco do arquivo, pois é um array só que guardado em lugares diferentes (dificuldade é achar o array). Na alocação por extensão precisa de uma lógica chatinha.

Tem que fazer uma “continha”, ver quantos blocos tem o pedaço contínuo e ir somando. Para evitar isso existe mais um campo na estrutura de controle no Windows que se chama número do bloco do arquivo, ou seja, o bloco que está no lugar nove é o quarto bloco do arquivo.

Qual é a alocação melhor?

Depende do arquivo, se ele for contínuo a alocação por extensão, que terá um único registro de controle, é melhor. Quando o arquivo é muito fragmentado a alocação indexada é melhor, pois será um único array com, por exemplo, 8 números. Já na por extensão terá uma estrutura com são guardados 18 números.

Numa nova versão do Linux (ext4) ele usa uma mistura das alocações, se ele for continuo com 4 pedaços contínuos usa a estrutura do Windows, se for mais fragmentado do Unix.

Arquivo Esparso

Fd = open (“A.dad”,o_wronly); -> somente escrita

Lseek(fd, 2048, Seek_end(onde começa)) -> Muda a posição atual para 2048 após o fim

Write (fd, &var, 1024);

Na hora da alocação como fica os buracos do arquivo? Na alocação indexada, no local dos blocos que estão com 0, o seu local no array recebe 0, indicando assim que o arquivo é esparso > 3 4 5 8 0 0 10

Na alocação por extensão fica indicado no número de bloco no arquivo, vai haver uma diferença nos números na estrutura. Isso é perceptivo quando somamos a quantidade de bloco com o número do bloco no arquivo e esse valor deve ser o valor do nó de bloco no arquivo no registro seguinte-> Vai do 4º bloco para o 7ª

Na alocação encadeada não permite arquivo esparso, pois não se pode apontar para o próximo bloco não esparso do arquivo, pois o bloco que seria o buraco não tem em disco, sendo assim, não tem o campo do próximo.

Controle dos Blocos Livres – Página 12

Bloco livre é o oposto de bloco alocado. Da para saber se um bloco é livre por exclusão, ver se ele não faz parte de nenhum arquivo, mas o desempenho é ruim, teria que ver a estrutura de controle de alocação de todos os arquivos de um volume. Para evitar isso existem estruturas de controle para dizer se o bloco está livre ou não

Vetor/bitmap/mapa de bites de bocós livre (mais simples) – Página 12 - Linux e Windows

É um vetor que cada posição corresponde a um bloco do volume. Tendo uma convenção que diz que 0 é o bloco livre e o 1 o bloco ocupado. E conforme o uso o vetor vai sendo alterado/preenchido.

Volume: 1TB
Bloco: 1KB

Quantidade de blocos de um volume = 1TB/1KB = 1G blocos

Vetor de bites = 1G bites = 128MB -> Não é grande, é relativamente pequeno

O espaço gasto pelo o vetor de bytes vai ser sempre 128 Mbytes, não importa se os blocos estão sendo usados ou não. > Espaço gasto com o controle é fixo

Lista de blocos livres – Lembra bloco de indireção

É uma lista que guarda o número dos blocos livres e, parecido com a ideia de blocos de idnireção, é guardada em um bloco livre, que passa a ser bloco de controle. Caso não seja suficiente guardar todos os blocos livre em um único bloco de controle é criado outro bloco de controle e o último byte é o número do próximo bloco de controle e no último é posto 0 para indicar que não tem próximo.> Não há um bloco de dupla indireção ou tripla indireção > Lista encadeada de bloco de controle

O primeiro bloco do volume é sempre um bloco de controle, chamado no unix de superbloco, diz quem é o número do primeiro bloco de controle da lista de blocos livres.

A eficiência dessa solução é pior, pois não há garantia que essa lista esteja em ordem, sendo assim, para saber se se um bloco tá livre precisa ler uma ou mais listas até encontrar o bloco ou não. No vetor de bytes é só ir à posição do vetor e vai saber se tá ocupado ou não.

A vantagem dessa solução é que ela ocupa um espaço variável, pois depende de quantos blocos de controle vai precisar > Espaço gasto com o controle é variável, depende da quantidade de blocos de controle necessários para guardar a lista.

Sendo assim, essa estrutura é econômica quando o disco está quase cheio, então precisa de poucos blocos de controles ou até um para guardá-los. Mas isso já perdeu o sentido atualmente, pois os discos não enchem com tanta facilidade. > Implementada na primeira versão do Unix

Alocação com FAT

Na FAT é a estrutura que controla alocação, lista encadeada de vários arquivos que estão no volume e os blocos livres, sendo os blocos livres são sinalizados com 0. > Muito grande e ocupa muita memória

Inconsistência de bloco – Página 12 verso

É o problema de se usar por exclusão para saber se um bloco está livre, pois quando há duas formas de se obter/gerar não há garantias de que elas retornem a

mesma informação. Ou seja, pode ocorrer algum problema que ocasione nessas informações estarem diferentes e gere uma inconsistência. > Formais normais estudadas em BD evitam essa inconsistência

1 Estruturas de controle de blocos livres diz que um certo bloco está livre, porém esse bloco está presente na estrutura de controle de alocação de um arquivo

Na hora que se cria um arquivo novo o SO pode usar esse bloco e na hora que salvar vai sobrescrever o conteúdo que é de outro arquivo.

Ex: incluiu novo bloco e antes de passar pro bitmap o pc apaga

2 Estrutura de controle de blocos livres diz que um certo bloco **não** está livre, porém esse bloco **não** está presente na estrutura de controle de alocação um arquivo (bloco perdido)

Ex: Criou um arquivo, alterou o bitmap, pc apaga e não alterou o inode do novo arquivo no disco

3 Um certo bloco está presente na estrutura de controle de alocação de mais de um arquivo – consequência da 1

Por que ocorre a inconsistência? Pois quando um arquivo é alterado é necessário mexer em 2 estruturas, na estrutura de controle de blocos livres e na estrutura de controle de alocação. Essas estruturas estão guardadas em locais diferentes no disco então pode ocorrer que na alteração você salvou em uma estrutura, mas acabou a luz e não salvou na outra e quando a máquina reiniciar vai estar inconsistente.

Inconsistência de arquivo (Inconsistência de link): Existe o contador de links no INODE e existem os diretórios que referenciam o INODE, logo, se tem dois diretórios referenciando o INODE o contador de link tem que marcar 2. Caso ele não marque existe uma inconsistência, isso ocorre pois quando se cria um novo link para um arquivo tem que alterar duas estruturas de dados (diretório e contador de link do arquivo), caso algo ruim aconteça na hora de mexer numa estrutura pode gerar inconsistência > não é tão grave, pois não se costuma criar links para arquivo, é mais fácil as inconsistências de bloco, pois é mais fácil criar ou apagar um arquivo

Correção das inconsistências

1 Programa buscador e corretor de inconsistência (antigo): A primeira vez que se usa um volume há uma marcação que indica que ele está em uso e quando a máquina é desligada corretamente essa marcação é desfeita. Quando há o desligamento errado e a máquina é ligada as marcações ainda estão lá, indicando que pode ter inconsistência e ai é iniciado o programa buscador de inconsistência (Unix fsch e Windows chkdsk). Eles procuram as inconsistências e as corrigem.

O problema é que para um volume grande com vários arquivos, essa busca pode demorar muito tempo, o que é um problema, visto que nenhum outro programa pode estar rodando junto com ele, pois se um programa estiver rodando ele pode estar alterando um arquivo que gera uma inconsistência temporária, por exemplo alterou o inode e não alterou o bitmap ainda. Sendo assim, pode ser que o buscador encontre essas inconsistências temporária e as marque.

Uso do conceito de transação (nova - atual): Ou nada é feito ou tudo é salvo. Vai logando a transação e depois esse log é varrido para ver se tem transações em aberto é feito o rollback. Isso é muito rápido > SGBD> Conceito de atomicidade > Implementação da atomicidade nas alterações das estruturas de controle (Tudo é feito ou nada é feito)

Desempenho do uso de arquivos – Página 13

Está relacionado com o fato de os arquivos estarem em disco, seja disco magnético ou SSD, que são muito mais lentos que a CPU.

Leitura de byte em byte

Chat c;

Int i, fd

Fd = open(“dados.dad”,O_RDONLY);

For (i=0;i<1024;i++){

Read(fd, &c,1 (quantidade de bytes a ler)) > Passa a variável c por referência, então o SO vai pegar o conteúdo do primeiro byte (byte apontado pela variável corrente), vai copiar para variável c e avançar a variável corrente para apontar para o segundo byte

//faço algo com c}

Quantas vezes, dentro do loop, o disco sofrerá uma leitura?

O loop ocorre 1024 vezes, ou seja, existem 1024 chamadas de leitura. Porém precisamos lembrar que existem unidade de alocação e leitura que são os blocos. Então, no primeiro loop é colocado o bloco inteiro em memória para ler o primeiro byte, e ele não é descartado por um tempo, pois outro ou o mesmo processo pode fazer uma leitura que pegue conteúdo desse bloco que já está em memória, sendo assim, não precisa ler do disco de novo. Sendo assim, 1vez, se for maior que 1024 (tamanho do bloco) e o tamanho do bloco for esse (1024) vai ser necessário mais de uma vez > SO mantém em memória os blocos recentemente lidos e escritos de forma que um uso futuro dele não precise ler o bloco do disco.

Cada bloco que é lido do disco vai para a memória, na parte do SO, que tem uma parte que guarda os conteúdos dos blocos lidos do disco. Essa parte se chama cache de disco do SO. Existe outra cache de disco do próprio disco (Hardware), ela põe na memória interna do próprio disco o conteúdo do disco. Quando a cabeça passa por cima do setor, por mais que não seja o setor desejado, ele aproveita e lê o conteúdo para a memória interna do disco. Então, ele já vai estar em memória, se precisar ler de novo ele já vai estar na memória interna do disco e já volta direto para o SO mais rápido (não precisa passar de novo passar no setor para ler). > Cache do SO é muito maior, ela visa aumentar o desempenho do uso de arquivo.

Chat c;

Int i, fd

Fd = open(“dados.dad”,O_WRONLY); // Quando o arquivo é aberto a posição corrente aponta para o primeiro byte do arquivo

For (i=0;i<1024;i++){

Read(fd, &c,1 (quantidade de bytes a ler)) > SO vai alterar o primeiro byte do arquivo (posição corrente) e vai avançar a posição corrente para byte seguinte, ou seja, vai apontar para o próximo byte não lido.

//faço algum cálculo e atribuo à variável c

Write (fd, &v,1);}

Quantas vezes, dentro do loop, o disco será escrito?

Quando se manda alterar o primeiro byte do arquivo, só será alterar um byte. Então o SO lê para a memória o primeiro bloco do arquivo e em memória e ele manda alterar somente o primeiro byte do bloco. Em seguida vai mandar salva o bloco ou não? Depende de como o SO quer implementar. (Write Through – Através) Existe a opção de mandar salvar em disco, nesse caso o disco será escrito 1024 vezes. (Write Back – Costas) A outra opção é não salvar o bloco toda hora e deixar acumular na cache de disco, e mais tarde se escreve no disco, mas quando?

Existe duas opções, a primeira é quanto a cache está cheia e precisa escolher um bloco vítima. Nesse caso da para usar LRU (bloco que foi usado a mais tempo no passado), pois o uso dele significa que algum processo pede para ler ou escrever um conteúdo que tá no bloco e o SO pode manter a organização na cache de disco que reflete uma organização LRU, ou seja, o cara do fim da fila é o que foi usado a mais tempo> (parecido com a página vítima que já foi vítima – foi alterada salva e se não foi, não salva) > não funciona na página porque um leitura não gera nada, nem interrupção, logo o SO não consegue manter uma organização. O problema é que a cache pode ser grande e, sendo assim, demorar muito para salvar o bloco, o que não é bom.

A segunda opção é temporal, de tempo em tempo os blocos alterados somente na cache de disco são salvos no disco. No Unix é de até 30 em 30 segundos, no Windows são poucos segundos.

Cache Write Through x cache Write Back

A desvantagem da primeira é a perda de desempenho, se se salva toda hora é preciso gastar tempo para isso. Caso deixe acumular vai salvar uma única vez. A desvantagem do segundo é armazenar várias coisas na cache o ocorrer um apagão ou similar e se perder tudo. > No Linux é fácil de ver, edita no editor, salva, fecha o programa e puxa o fio e quando você for ver não foi salvo, no Windows é mais difícil devido a opção temporal.

Conceito de cache – Página 14

Usar um meio de armazenamento mais rápido, nos casos anteriores é a memória, para guardar um conteúdo que está em um meio de armazenamento mais lento, disco magnético, com o propósito de aumentar o desempenho.

Exemplo: Quando não se tem uma conexão de rede não muito boa e se está se consultando um site pela primeira vez, se demora para aparecer a página porque precisa salvar todo o conteúdo que está em um servidor remoto para exibir. Porém, quando esse conteúdo vem para o browser ele vai guardar no disco local e caso tente-se abrir a mesma página num futuro próximo o browser não precisa pegar o conteúdo novamente, pois ele já o salvou. Ele faz somente outra chamada para saber se mudou algo, se não tiver não precisa pegar nada e se mudou se pega somente o que foi alterado. Ou seja estamos usando um mecanismo de armazenamento mais rápido, o disco local (magnético) em vez de usar o armazenamento mais lento o servidor remoto.

Outro exemplo é dos tipos de memória que a CPU pode consultar, a memória principal é bem lenta em relação a CPU, para evitar a CPU ficar parada existe uma hierarquia de memória. Tem a cache, que é muito mais rápida e evita que se vá na memória principal e se gaste mais tempo.

Cache de disco X Memória Virtual (Paginação usando o disco) – Página 14

A cache de disco usa memória para guardar conteúdo que estava originalmente no disco com o propósito de aumentar o desempenho. Já a memória virtual, quando a memória está cheia ela salva páginas no disco, ou seja, ela usa o disco para guardar conteúdo que originalmente ficam na memória (aumentar páginas). Cache de disco você usa a memória para simular o disco, já na virtual você usa o disco para guardar o conteúdo que estava na memória. Ou seja, se usa o disco para simular a memória.

No sentido mais prático, existente um gasto de memória dentro do SO para guardar a cache de disco, por exemplo 1GB e os outro 7GB são para armazenar o processo. Caso encha esses 7GB precisa começar a liberar páginas em memória e as colocá-la em disco, mas já se está gastando 1GB com a cache de disco. Então, faz sentido manter esse 1GB para cache de disco sendo que a memória dos processos está ficando cheia e começa a guardar coisas em disco? A típica resposta é não, pois é mais interessante gastar memória para guardar páginas, pois para guardar blocos do disco é somente vantajoso se você voltar a usar o bloco no futuro, mas não é garantido usar.

No entanto, não é bom deixar a cache de disco muito pequena, pois vai gerar uma dificuldade para usar os blocos do disco. Quando se faz uma leitura em vez de armazenar na cache ela vai ser apagada e quando ser outra parte vai precisar ir à memória de novo. Então ela não pode ficar muito pequena, pois para usos mais simples vai precisar ir buscar bloco do disco toda hora.

Hoje em dia nos tomos uma “gerência unificada de memória”. Então tanto a cache de disco e a memória do processo são tratadas da mesma forma e tudo passa a virar página. Então a escolha vira uma escolha de página vítima.

Arquivo mapeado em memória > Arquivo como se estivesse em memória

É uma outra forma de usar arquivos sem usar as chamadas tradicionais de alteração e consulta. A abertura de arquivo segue existindo, mas se quiser consultar pode-se mapear o arquivo em memória.

Int fd;

Fd = open(“dados.dad”,O_RDONLY);

Char *p; // irá apontar para o endereço inicial do mapeamento

P = mmap(fd, ….); > Mapea um arquivo em memória e retorna o endereço de onde o arquivo está mapeado na memória do processo, mas mapear não lê o arquivo para a memória

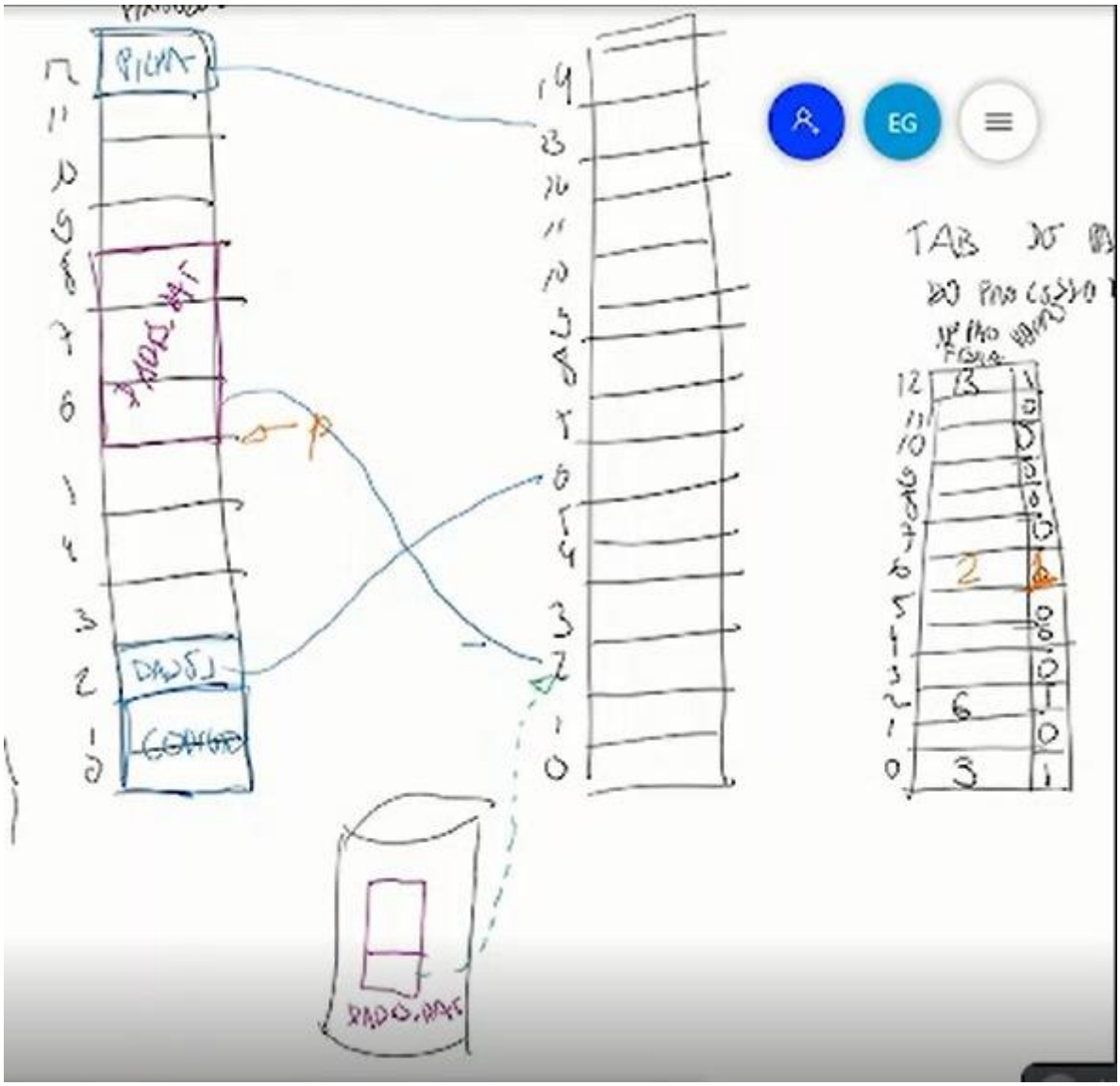
C = [p0]; // como se fosse uma chamada read e por o endereço na variável c > faz carregar na memória a página

P[2] = ‘x’; > Está alterando o terceiro byte do arquivo, equivale a chamada write

Lembra muito o carregamento sob demanda, onde as páginas executáveis vão ser carregas conforme necessário, mas ele é feito para um arquivo qualquer. Então quando se mapeia ele para a memória é como se o arquivo estivesse ocupando uma sequência de páginas na memória lógica, mas na realidade não se leu nada, pois elas seguem inválidas na tabela de páginas do processo, pois ainda não foram lidas.

Então se usa o arquivo como um array e pode-se consultar ou alterar uma posição do vetor. E quando se usar um byte vai gerar uma interrupção, pois a página está inválida, mas o processo não é abortado porque é para ter um arquivo mapeado, pois é válido, mas ainda não se carregou o arquivo ainda. Então será pego do disco os primeiros 4kilobytes e joga na página física livre e em seguida ele vai fazer a primeira página lógica do arquivo apontar para a posição física. Depois a instrução é reiniciada e é possível carregar o primeiro byte para c > Não carrega nada e só vai carregando sob demanda e com ele em memória se pode usar como se ele estivesse em memória e se usa como array (não se usa as chamadas de read)

Não existe um comando de salvar em disco, esse cara é como se fosse um arquivo em memória. Inicialmente ele só altera em memória e mais tarde esse conteúdo vai ser salvo em disco > Cache write back. > Se aletrar o conteúdo VAI SALVAR NO DISCO eventualmente > Só entra em memória quando se usa



NA chamada convencional tem uma grande deficiência na leitura de grandes partes que um arquivo grande.

Chat a[1000000000];

Int fd

Fd = open(“b.dad”,O_RDONLY);

Read(fd, a,1000000000 (quantidade de bytes a ler))

Quando manda ler um pedaço para o arquivo se pega o bloco que você quer e põem na cache de disco e depois transferir para a variável a que aponta para uma área na memória do processo na área dos dados. Ou seja, ocupando 100MG na cache do disco e na área de dados com o mesmo conteúdo, que seria um desperdício de memória. > Acontece nas chamadas tradicionais ocupa a cache de disco e a variável

No mapeamento em memória não tem essa duplicidade, pois quando você mapeia ele não vai para a cache de disco.

Caso um processo use um arquivo mapeado e outro as mesmas só que na forma convencional não vão aparecer duplicados, ambas as memórias do processo vão apontar para o mesmo lugar na página física. > Isso trata a cache de disco como se posse página

O critério para saída é através de página vítima NRU para sair da memória física > Não precisa encolher a cache de disco, as que estão sem uso vão ser vítimas