

Texto do CFRED

Dividindo a memória em 8, poderíamos ter 8 processos rodando ao mesmo tempo, cada um em seu espaço. Das oito, uma é reservada para armazenar informações (status) dos processos. Sendo que na CPU, o único elemento que guarda estados são os registradores.

Quando um programa espera os 27 mil anos, um “supervisor” irá copiar os dados dos registradores no bloco de memória reservado para ele e escolherá um processo dos que estão em espera na memória, carrega seus registradores e volta a ser executado, sendo um dos registradores o PC do processo.

O supervisor existe para que o programa não precise ficar os 27 mil anos ocupados perguntando se o dado está pronto, fazendo uma espera ocupada.

Entre os registradores, um que acompanha o fluxo de execução é o da pilha, que aponta sempre para o seu topo. As informações podem ser tiradas do reg e adicionadas na pilha com o PUSH e lidas e escritas no reg com o POP.

Quando uma função é chamada, usamos a instrução CALL ADD que coloca o endereço atual na pila e muda o endereço para ADD, depois executa o código da função, quando a mesma é finalizada, é utilizada a instrução RET que traz de volta o endereço anterior

A instrução CALL ADD pode ser invocada via hardware a partir da mudança de valores de pinos do processador. Se houver uma mudança no pino 1, o processador faz um CALL 100, por exemplo.

Associando o pino 1 ao teclado, sempre que uma tecla for pressionada, ocorre uma mudança de estado no pino e a CALL 100 é chamada. O registrador PC é preservado para não apagar outros dados (PUSH) e o processo de ler o teclado salva todos os seus registradores em um buffer, para depois retornar o estado dos registradores como antes (POP)

Capítulo 2

Processos

Chaveamento de processos é a troca de processos a cada período de tempo

Cada processo possui o seu PC lógico, e quando é a sua vez de entrar em execução, o seu valor é carregado para o processador físico.

Há quatro eventos principais que criam processos

1. Início do sistema
2. Criação de um processo a partir de outro
3. Requisição do usuário
4. Início de uma tarefa em lote

Daemons são processos que ficam no background e é ativado para lidar com serviços requisitados

fork: Cria um processo filho idêntico ao pai, cada um com seu espaço de memória

Há quatro principais eventos que encerram um processo:

1. Saída normal (voluntária).
2. Saída por erro (voluntária).
3. Erro fatal (involuntário).
4. Cancelamento por um outro processo (involuntário).

Um processo possui três estados principais:

1. Em execução: Está usando a CPU
2. Parado: Está esperando a sua vez de usar a CPU
3. Bloqueado: Esperando um evento acontecer (Mesmo com a CPU liberada, não pode fazer nada)

O processo vai de 1 para 2 quando seu tempo de execução acaba, e de 2 para 1 quando todos os outros processos já usaram a CPU.

Vai de 1 para 3 quando é identificado que ele necessita de um evento para que ele possa prosseguir. E vai de 3 para 2 quando o seu evento ocorre.

O sistema operacional implementa uma tabela de processos com informações sobre o estado do processo, seu PC, SP, alocação de memória, estado dos arquivos abertos, informações de escalonamento e tudo que é necessário para salvar e carregar durante as transições de estado do processo. Em geral, a tabela é dividida em gerenciamento de processo, memória e arquivo.

Processo de tratamento de interrupções:

1. O hardware empilha o PC e carrega novo
2. O assembly salva os registradores e configura uma nova pilha
3. O serviço de interrupção em C lê e armazena a entrada
4. O escalonador decide o próximo processo a ser executado
5. O procedimento em C retorna para o código assembly, que inicia o novo processo

Quanto maior o número de entradas e saídas de um programa, maior deve ser o nível de multiprogramação a fim de evitar que a CPU seja desperdiçada

Threads

São processos de um mesmo programa que compartilham o mesmo endereço de memória e executam simultaneamente.

Criar uma thread é mais rápido do que um processo, pois não possui recursos associados. Por isso as threads também são chamadas de LWP (Low Weight Process, ou processo de baixo peso)

Threads são muito úteis quando há a chamada de sistema bloqueantes, pois somente a thread em questão é bloqueada e as outras podem continuar o seu processamento. Exceto quando a chamada bloqueia todo o processo.

Chamadas de sistema bloqueantes: São chamadas de sistemas que interrompem o fluxo de execução até que um processamento seja concluído, como a solicitação de um arquivo que está no disco

Threads compartilham um espaço de endereçamento, variáveis globais, arquivos abertos, processos filhos, alarmes, sinais, etc. Porém, cada um possui o seu próprio PC, registradores, pilha e estado (Os mesmos dos processos, as transições também) .

As threads possuem uma tabela de threads, que contém informações sobre as threads do processo, mantendo informações para reiniciar uma thread quando ela vai para o estado de bloqueada ou espera.

Quando um thread é iniciado, só é passada a sua rotina. O seu espaço de memória não pode ser informado, pois ele é o mesmo do processo.

Thread de usuário

São threads que os programadores criam.

Cada processo possui a sua própria tabela de threads, gerenciada pelo sistema de tempo de execução.

Threads de usuário podem implementar o seu próprio escalonador.

Uma chamada de sistema bloqueante em uma thread bloqueia todos os outros threads

Um thread de usuário é identificado e linkado numa thread de kernel

Thread de núcleo (kernel)

Um thread de kernel é mais rápido que um thread de usuário, pois ele é de fato reconhecido, porém a sua criação e manutenção é muito mais difícil e demorada. Ademais, a sua tabela fica na parte do núcleo.

Quando um thread de núcleo quer criar um novo thread ou destruir um existente, ele faz uma chamada no núcleo, que realiza a tarefa atualizando a tabela de threads no núcleo. Este processo é muito custoso, então muitos threads são reaproveitados.

Quando há uma chamada de sistema bloqueante, somente a thread é bloqueada, o processo continua.

Threads híbridas

É quando uma thread de núcleo identifica as threads de usuário e se conecta a elas, utilizando o melhor dos dois mundos: A rápida criação de uma thread de usuário e o não bloqueio de uma thread de núcleo.

Escalonador de threads

Quando ocorre uma chamada de sistema bloqueante, o sistema de tempo de execução é informado e ele reescala os threads colocando o atual como bloqueado. Dessa forma, quando o processo não ficar mais bloqueado, ele é reinicializado ou colocado na lista de pronto

Se ocorre uma interrupção no nível de hardware enquanto há uma thread de usuário, a CPU vai para o modo núcleo e depois de finalizar a interrupção, o thread volta para o estado em que estava. Mas se o thread estava interessado na

interrupção, ele vai para uma CPU virtual e o sistema decide qual processo escalonar para aquela cpu

Outras informações sobre threads

O modelo de processo é baseado em dois conceitos independentes: agrupamento de recursos e execução

Processos são usados para agrupar recursos e as threads são usadas para escalar sobre a execução da CPU.

- `thread_create`: Inicia um novo thread
- `thread_exit`: Um thread finaliza seu trabalho e deixa de ser chamado
- `thread_join`: Um thread espera um outro específico terminar
- `thread_yield`: Faz com que uma thread desista, voluntariamente, da CPU

Hoje em dia, o windows possui um grande suporte ao pthread e as threads são, em geral, híbridas. Dessa forma, uma thread de usuário que foi bloqueada não bloqueará as outras.

Comunicação entre processos

Condições de corrida são situações quando dois ou mais processos, em paralelo, estão acessando uma região crítica ao mesmo tempo. Dessa forma, a ordem de execução dos processos concorrentes alteram a computação, já que não podemos prever quando o escalonador de processos decidirá que o tempo de acesso à CPU de uma thread acabou e outra deverá entrar em execução.

Exclusão mútua é uma forma de garantir que nenhum processo irá acessar uma variável ou arquivo compartilhado que já está em uso por outro processo, pois só um processo por vez tem acesso à região crítica.

Região crítica é uma parte do programa que possui acesso à memória compartilhada por um processo e não pode ser acessado por outro. Ela possui 4 leis para uma boa implementação:

1. Dois processos não podem estar ao mesmo tempo em suas regiões críticas
2. Nada pode ser afirmado sobre a velocidade ou número de CPUs
3. Um processo que não está na região crítica não pode bloquear outros processos
4. Não deve haver uma espera eterna para acessar a região crítica

Espera ociosa: é quando há um laço de repetição que fica verificando o estado de uma variável até que ela atinja o valor desejado.

2.3.3 Maneiras de implementar a exclusão mútua com espera ociosa:

Desabilitar as interrupções, e consequentemente o chaveamento, não é uma boa ideia pois pode ser que o processo não reabilite (Quebra a regra 4) ou pode acontecer de bloquear somente para uma CPU (Quebra a regra 2). Assim, outras acessam a região crítica (Quebra a regra 1)

Usar uma **variável de chave** não é uma boa ideia, pois pode ocorrer de um processo A ler que não possui ninguém na região crítica, e, na hora de mudar o estado da variável após ter entrado na mesma, o escalonador decide que o seu

tempo acabou. Assim, o processo B também vê que não há ninguém, acessa e muda a chave. Assim teremos dois processos na região crítica. (Quebra a regra 1)

O **chaveamento obrigatório** faz uma espera ocupada entre os processos, mas pode ocorrer de um processo A ter acesso à região crítica, sair, dar acesso ao B e realizar o seu código na região não crítica e voltar para a espera ocupada. Por outro lado, o processo B irá entrar, realizar suas operações e dar acesso ao A, depois entrar na sua região não crítica, que é muito grande. O processo A vai para a região crítica, sai dá acesso ao B e volta para a espera ocupada, mas o processo B ainda está na região não crítica. Logo, a regra 3 é descomprida. Caso a thread B seja finalizada, ela ainda irá quebrar a regra 4, pois o A nunca mais entrará na região crítica.

```
while (TRUE) {  
    while (turn !=0)          /* laço */ ;  
    critical_region( );  
    turn = 1;  
    noncritical_region( );  
}
```

(a)

```
while (TRUE) {  
    while (turn !=1)          /* laço */ ;  
    critical_region( );  
    turn = 0;  
    noncritical_region( );  
}
```

(b)

```

#define FALSE 0
#define TRUE 1
#define N      2                /* número de processos */

int turn;                        /* de quem é a vez? */
int interested[N];              /* todos os valores 0 (FALSE) */

void enter_region(int process);  /* processo é 0 ou 1 */
{
    int other;                  /* número de outro processo */

    other = 1 - process;        /* o oposto do processo */
    interested[process] = TRUE; /* mostra que você está interessado */
    turn = process;             /* altera o valor de turn */
    while (turn == process && interested[other] == TRUE) /* comando nulo */ ;
}

void leave_region(int process)   /* processo: quem está saindo */
{
    interested[process] = FALSE; /* indica a saída da região crítica */
}

```

Figura 2.19 A solução de Peterson para implementar a exclusão mútua.

O programa começa e os processos 0 e 1 não estão interessados. O processo 0 fica interessado e chama a função `enter_region` antes de executar o código da entrada da zona crítica. Coloca o seu valor no vetor de interesse como verdadeiro e coloca como `turn` o seu número. Depois disso, ele sai da função e vai para a zona crítica caso o valor de turno não seja o seu (isso indica que o outro processo não está na área crítica, já que ele mudou uma variável de controle) ou caso o outro processo não esteja mais interessado (A função `leave_region` foi chamada)

A instrução TSL lê o conteúdo da memória no registrador `RX` que possui uma variável compartilhada chamada `lock`, quando ela está em 0, qualquer um pode acessar a região crítica. Quando vai acessar, muda-se o valor de `lock` para 1 e faz um bloqueio de barramento, impedindo qualquer outro processador de acessá-la

Antes de entrar na região crítica, o processo chama `enter_region` e fica em uma espera ociosa até que a trava seja liberada. Quando for liberado, retorna para onde foi chamado, executa o código, vai para `leave_region` e continua o seu trabalho

```

enter_region:
    TSL REGISTER,LOCK    | copia lock para o registrador e põe lock em 1
    CMP REGISTER,#0      | lock valia zero?
    JNE enter_region     | se fosse diferente de zero, lock estaria ligado, portanto, continue no laço de repetição
    RET                  | retorna a quem chamou; entrou na região crítica

leave_region:
    MOVE LOCK,#0         | coloque 0 em lock
    RET                  | retorna a quem chamou

```

2.3.4 Dormir e acordar

Sleep faz com que o processo que a chamou fique suspenso

Wakeup possui um parâmetro para acordar o processo que está dormindo

O problema do produtor-consumidor

Dois processos compartilham o mesmo buffer de tamanho fixo. Um coloca informação (produtor) e o outro tira (consumidor).

Se o produtor quiser colocar mais itens do que cabe no buffer, ele dorme. O mesmo ocorre para o consumidor se quiser tirar algo quando não tem nada. E ficam assim até que possam fazer algo.

No código, N é a capacidade máxima de itens no buffer e count é a quantidade de itens atual.

Na função producer, há um loop infinito e dentro dele um novo item é gerado, depois verifica-se se o buffer está cheio, caso esteja, ele vai dormir até que o processo consumidor o acorde. Quando for acordado, significa que o buffer não está mais cheio, então adiciona o item e incrementa o contador. Depois verifica se a quantidade de itens do buffer é 1.

```

#define N 100                                /* número de lugares no buffer */
int count = 0;                               /* número de itens no buffer */

void producer(void)
{
    int item;

    while (TRUE) {                            /* repita para sempre */
        item = produce_item();               /* gera o próximo item */
        if (count == N) sleep();             /* se o buffer estiver cheio, vá dormir */
        insert_item(item);                   /* ponha um item no buffer */
        count = count + 1;                   /* incremente o contador de itens no buffer */
        if (count == 1) wakeup(consumer);    /* o buffer estava vazio? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                            /* repita para sempre */
        if (count == 0) sleep();             /* se o buffer estiver cheio, vá dormir */
        item = remove_item();                /* retire o item do buffer */
        count = count - 1;                   /* decresça de um o contador de itens no buffer */
        if (count == N - 1) wakeup(producer); /* o buffer estava cheio? */
        consume_item(item);                  /* imprima o item */
    }
}

```

O código ainda possui uma condição de corrida que pode infringir as leis 3 e 4 da região crítica, pois o count pode ser acessado a qualquer momento.

O consumidor leu a variável count, que está zerado, e logo em seguida o escalonador decide que é a vez do produtor usar a CPU. Ele produz o item, coloca no buffer, incrementa o contador que vai de zero para um e, por isso acorda o consumidor, depois volta para o início do loop.

O escalonador volta para o consumidor, ele parou na verificação do contador, como estava em zero, então o próximo passo é dormir. Como ele não estava dormindo antes, o acordar do produtor foi ignorado.

O escalonador volta para o produtor, que produz mais itens e, como o contador é maior que 1, não acorda o consumidor. Então produz até encher o buffer, vai dormir e nenhum dos dois é acordado, gerando uma espera infinita

Uma solução para esse problema seria adicionar um bit verificador de sinal de envio de acordo. Dessa forma, se for enviado e o processo ainda estiver acordado, então ele não dormirá, mas pode não funcionar para mais que dois processos.

2.3.5 Semáforos

Uma variável inteira conta quantos sinais de acordar devem ser enviados.

Se o semáforo for zero, não há recursos livres, caso contrário, o recurso está livre, e se há processos que estão dormindo, mas precisam deste recurso, então ele deve ser acordado.

O semáforo pode conter o valor zero se nenhum sinal de acordar foi salvo ou algum valor positivo caso houvesse sinais de acordar pendentes.

Down: Verifica se seu valor é maior que zero. Se for, gasta um sinal de acordar (decrementa) e prossegue. Se o valor é zero, o sistema é colocado para dormir sem terminar o down.

Verificar o valor, alterá-lo e talvez dormir são tarefas executadas como uma só de forma indivisível. Assim, nenhum processo acessa o semáforo até que a operação acabe ou seja bloqueada, impedindo a condição de corrida.

Up: Incrementa o valor de um semáforo. Se um ou mais processos estiverem dormindo, um deles é escolhido para terminar o seu down. Quando há um up com processos dormindo, o semáforo continua em zero.

Resolvendo o problema do produtor-consumidor com semáforos

Full: Inicia com zero e conta o número de lugares preenchidos

empty: Inicia com o número de lugares no buffer e conta quantos lugares estão vazios

mutex (mutual exclusion): Impede o acesso ao buffer do consumidor e produtor ao mesmo tempo e inicia com 1, pois é usado por dois ou mais processos.

Semáforos que iniciam com um e são usados por 2 ou mais processos são chamados de semáforos binários.

Cada processo faz um down antes de entrar na região crítica e um up logo após sair.

Cada dispositivo de E/S tem um semáforo que inicia com zero. Assim que ele é inicializado, ocorre um down e o processo é bloqueado. E é colocado em pronto quando ocorre uma interrupção e o processo recebe um up.

```
#define N 100                                     /* número de lugares no buffer */
typedef int semaphore;                             /* semáforos são um tipo especial de int */
semaphore mutex = 1;                               /* controla o acesso à região crítica */
semaphore empty = N;                               /* conta os lugares vazios no buffer */
semaphore full = 0;                                /* conta os lugares preenchidos no buffer */

void producer(void)
{
    int item;

    while (TRUE) {                                /* TRUE é a constante 1 */
        item = produce_item();                    /* gera algo para pôr no buffer */
        down(&empty);                             /* decresce o contador empty */
        down(&mutex);                             /* entra na região crítica */
        insert_item(item);                        /* põe novo item no buffer */
        up(&mutex);                                /* sai da região crítica */
        up(&full);                                 /* incrementa o contador de lugares preenchidos */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                                /* laço infinito */
        down(&full);                               /* decresce o contador full */
        down(&mutex);                             /* entra na região crítica */
        item = remove_item();                     /* pega item do buffer */
        up(&mutex);                                /* sai da região crítica */
        up(&empty);                                /* incrementa o contador de lugares vazios */
        consume_item(item);                       /* faz algo com o item */
    }
}
```

Figura 2.23 O problema produtor–consumidor usando semáforos.

Para o produtor, primeiro verificamos se há locais vazios no buffer, se não houver, o processo ficará suspenso enquanto espera. Tendo espaço então ele altera o valor de mutex para já entrar na região crítica. Em seguida, já sai da mesma alterando o valor de mutex e incrementa em um o valor de full, pois há um novo item no buffer.

Já para o consumidor, primeiro verifica se há algum item no full, se houver, realiza um down para o full e o down para o mutex. Depois entra na região crítica e já saindo um up no mutex e, em sequência, um up no empty, pois um item foi consumido e há uma nova vaga livre.

Caso os downs estivessem trocados, poderíamos ter problemas, pois, no produtor, verificamos se o buffer não está cheio para só depois verificar se pode entrar na região crítica. Dessa forma, caso pudesse entrar na região crítica, mas o buffer estivesse cheio, então ficaria preso esperando que algum item fosse consumido, mas o mutex estaria em zero, então o consumidor não poderia entrar na região crítica para consumir um item e liberar o produtor

2.3.6 Mutexes

Versão simplificada de semáforos para quando não precisa contar.

Mutex é uma variável que pode estar impedida (1) ou desimpedida (0).

Se um thread precisa de acessar a região crítica, chama a `mutex_lock`. Com o mutex desimpedido, o código entra na região crítica. Com o mutex impedido, o thread fica bloqueado até que o thread que está na região crítica chame o `mutex_unlock` para sair.

Se múltiplos threads estão bloqueados sobre o mutex, um é escolhido aleatoriamente para acessar a região crítica e ter acesso à trava

O código é muito semelhante ao do uso do TSL com a variável de trava, porém, neste, supondo uma implementação totalmente no espaço de usuário, o thread ficaria para sempre no loop infinito, pois sempre estaria testando e não há um escalonador de threads com relógio pré implementado, dessa forma, é necessário um `thread_yield` para que o thread atual libere o processamento para outro thread.

mutex_lock:	
TSL REGISTER,MUTEX	copia mutex para o registrador e atribui a ele o valor 1
CMP REGISTER,#0	o mutex era zero?
JZE ok	se era zero, o mutex estava desimpedido, portanto retorne
CALL thread_yield	o mutex está ocupado; escalone um outro thread
JMP mutex_lock	tente novamente mais tarde
ok: RET	retorna a quem chamou; entrou na região crítica
mutex_unlock:	
MOVE MUTEX,#0	coloca 0 em mutex
RET	retorna a quem chamou

O Thread deseja entrar na região crítica, então ele chama o mutex_lock, que verifica se o valor do mutex é zero, se for, retorna para onde foi chamado e entra na região crítica. Se não, dá o uso da CPU para outro thread e depois volta chamando novamente o mutex_lock, ficando em uma espera ocupada.

Quando for sair, vai para o mutex_unlock e coloca 0 no mutex, depois retorna para onde foi chamado.

2.3.7 Monitores

Semelhante aos semáforos, mas de mais alto nível e menos suscetível a erros do programador

Um monitor é uma coleção de rotinas, variáveis e estruturas de dados agrupadas

Processos podem chamar rotinas de um monitor quando quiserem, mas não podem acessar os dados internos do monitor quando forem chamados por fora dele.

Somente um processo pode estar ativo em um dado momento.

Os monitores são estruturas da linguagem, os compiladores entendem e os tratam de forma diferente.

Quando um processo chama uma rotina, verifica-se se há algum processo ativo no monitor. Caso tenha, o processo que chamou ficará suspenso até que outro processo deixe o monitor.

A exclusão mútua é implementada pelo compilador, desta forma, o programador só precisa converter as regiões críticas em rotinas do monitor.

wait: Quando uma rotina descobre que não pode prosseguir, executa o wait sobre uma variável condicional, bloqueando o processo que está chamando, permitindo que outro processo, anteriormente bloqueado, agora possa entrar.

signal: Para evitar que dois processos fiquem no monitor ao mesmo tempo, é enviado um signal para uma variável condicional para acordar outro processo. Sendo o signal o último comando da rotina, e se vários processos estão esperando esse signal, somente um deles é chamado pelo escalonador.

wait e signal são semelhantes a sleep e wakeup, porém se o wakeup enviar um sinal para alguém que não está dormindo, o mesmo será perdido. Dessa forma, o wait e o signal evitam esse problema permitindo que a operação wait seja terminada.

2.3.8 Troca de mensagens

send e receive são chamadas de sistema para comunicação entre processos. Se nenhuma mensagem estiver disponível, o receptor pode ficar bloqueado esperando ou pode retornar um erro.

Conectando dois processos pela rede, um pode mandar uma mensagem e o que recebeu pode enviar uma mensagem de confirmação. Mas pode ocorrer da confirmação não ser recebida, fazendo com que o emissor mande duas vezes. Dessa forma, muitas mensagens são enviadas com um número sequencial, assim, o processo pode verificar se já recebeu a mensagem e ignorá-la.

2.3.9 Barreiras

Barreiras impedem que algum processo avance enquanto há processos que ainda não estão prontos para avançar, bloqueando o mesmo.

Quando vários processos estão executando e atingem um ponto, eles executam a primitiva barrier, a partir de uma biblioteca, suspendendo o processo. E, assim que todos atingirem a mesma, os processamentos retornam

2.4 Escalonamento

Escolhe qual processo que está em pronto irá usar a CPU

2.4.1 Introdução ao escalonamento

A escolha dos processos deve ser bem feita, pois chavear é muito custoso.

Primeiro há um chaveamento de modo de usuário para modo de núcleo. O estado atual do processo é salvo e depois um novo processo é escolhido pela MMU (memory management unit), que carrega todo o processo.

Comportamento do processo

Quase todos os processos alternam entre o uso da CPU e E/S, podendo ser de espera quando eles ficam bloqueados ou quando se está escrevendo na RAM.

- Processos limitados pela CPU: Gastam a maior parte do tempo computando e às vezes esperam por E/S
- Processos limitados por E/S: Ficam a maior parte do tempo em espera e pouco tempo de fato computando.

Quando um processo acaba de ser executado, mas não há nenhum outro processo em pronto, o sistema operacional gera um processo ocioso.

Quando ocorre uma interrupção, o escalonador decide se irá executar o processo que acabou de ficar pronto, o que foi interrompido ou outro processo.

Alguns sistemas possuem relógios que permitem a separação de tempos.

- Algoritmo escalonador não preemptivo: Escolhe um processo e deixa executar até que seja bloqueado ou que libere a CPU voluntariamente
- Algoritmo escalonador preemptivo: Executa o processo por um período de tempo fixo. Necessita de um relógio para interromper o processo e a CPU volte para o escalonador

Os algoritmos de escalonamento são otimizados para três grandes usos:

- Lote: Não há usuários no terminal, algoritmos usados são não-preemptivos ou preemptivos com um longo intervalo e tempo para cada processo.
- Interativos: A preempção é necessária para que um processo não tome posse da CPU e caso um erro ocorra, não fique muito tempo no processo. Servidores também usam esse algoritmo.
- Tempo real: A preempção é, às vezes, desnecessária, pois os processos são feitos de forma a não serem executados por longos períodos.

Objetivos dos algoritmos sob algumas circunstâncias:

- Para todos os sistemas:
 - Justiça: Processos semelhantes devem ter tempos semelhantes
 - Política: Uma política estabelecida deve ser cumprida. Ex: cada processo deve durar 30s.

- Equilíbrio: Todas as partes do sistema devem ser ocupadas. Vale a pena para o lote ter processos limitados pela CPU e E/S sendo executados para manter tudo sendo utilizado
- Sistemas em lote:
 - Vazão: Maximizar o número de tarefas por hora
 - Tempo de retorno: Minimizar o tempo entre chamada e resposta
 - Utilização da CPU: manter a CPU ocupada o tempo todo
- Sistemas interativos:
 - Tempo de resposta: Responder o mais rapidamente às requisições para diminuí-lo
 - Proporcionalidade: satisfazer às expectativas dos usuários em relação ao tempo demorado por tarefas simples e complexas.
- Sistemas de tempo real:
 - Cumprimento de prazos: evitar a perda de dados. Ex: Um sistema que depende da temperatura não pode perder informações
 - Previsibilidade: Evitar a degradação da qualidade em sistemas multimídia

Nos sistemas em lote, caso tenhamos muitas tarefas curtas chegando e sendo executadas e poucas tarefas longas, a vazão pode ser muito alta e o tempo de retorno infinito.

2.4.2 Escalonamento em sistemas em lote

FIFO: algoritmo não preemptivo que forma uma fila e o primeiro a chegar tem acesso à CPU. Caso ele seja bloqueado, sairá da fila e será posto no final ao ficar pronto.

Tarefa mais curta primeiro (shortest job first - SJF): algoritmo não preemptivo que conhecendo o tempo de execução de cada tarefa coloca a mais rápida no começo. Uma tarefa muito grande pode morrer de inanição caso continuem a chegar somente tarefas menores do que ela.

Próximo de menor tempo restante: Versão preemptiva do anterior, mas caso uma tarefa nova chegue, ela é comparada com a que está em execução e seu tempo para finalização. Se for menor, ela é executada, caso contrário, continua a execução como está.

2.4.3 Escalonamento em sistemas interativos

Usados em computadores pessoais e servidores

Escalonamento por chaveamento circular: Todos os processos são igualmente importantes e cada processo possui um tempo (quantum) que ele pode executar. Caso não termine até o fim do seu tempo, a CPU vai para outro processo e ele vai pro fim da fila. Se ele é terminado antes ou bloqueado, a CPU chaveia outro processo.

Um quantum curto gera muitos chaveamentos e torna a CPU ineficiente, mas um quantum longo gera longos períodos de espera

Escalonamento por prioridades: Cada processo possui uma prioridade, e aquele com a maior será executado.

Um processo pode ter sua prioridade diminuída a cada tique do relógio, permitindo outros a serem executados.

Pode ser atribuído um tempo e, quando ele acabar, outro processo assume.

Processos que utilizam mais entradas e saídas podem ter uma prioridade maior, já que eles gastarão menos tempo usando a CPU e mais tempo esperando a E/S.

Um algoritmo usado é atribuir a prioridade como sendo $\text{quantum}/\text{tempo_usado}$.

Muitas vezes, processos são colocados em grupos de prioridade e, assim que um grupo finaliza, o próximo é executado. Pode ocorrer de um grupo mais baixo nunca ser executado, pois a prioridade dos de cima aumentou.

Filas múltiplas: Cria-se diferentes filas, cada uma com um certo número de quantuns disponíveis. Caso um processo utilize todos, ele é movido para a fila de cima que possui mais quantuns.

Próximo processo mais curto (shortest process next): Executa o processo cujo tempo de execução estimado seja o menor. Sendo este tempo calculado a partir dos tempos passados do processo, de forma que ele envelheça.

Escalonamento garantido: Cada processo terá um tempo de $1/n$ para ser executado, onde n é o número de processos.

Escalonamento por loteria: Quando ocorre um escalonamento, o processo a usar a CPU será escolhido aleatoriamente, sendo que processos mais importantes possuem maiores chances de serem escolhidos e processos podem dar suas chances para outros.

2.4.4 Escalonamento em sistemas de tempo real

Hardwares externos geram dados que o computador deve lidar em um intervalo de tempo.

- Tempo real crítico: Prazos absolutos que devem ser cumpridos sem erros
- Tempo real não crítico: Perdas e atrasos são indesejáveis, mas toleráveis

O programa é dividido em vários processos com comportamento conhecido, sendo que eles possuem uma vida curta e podem executar em menos de um segundo. Se um evento externo é detectado, o escalonador escalona os processos para cumprir os prazos.

2.5 Problemas clássicos de IPC

Problemas de sincronização, entre eles o jantar dos filósofos.

N filósofos estão em um jantar, mas para comerem precisam do garfo à esquerda e à direita. A ideia é fazer um algoritmo para sincronizar e permitir que um filósofo pense e coma sempre que possível.

```

#define N          5                /* número de filósofos */
#define LEFT      (i+N-1)%N        /* número do vizinho à esquerda de i */
#define RIGHT     (i+1)%N          /* número do vizinho à direita de i */
#define THINKING  0                /* o filósofo está pensando */
#define HUNGRY    1                /* o filósofo está tentando pegar garfos */
#define EATING    2                /* o filósofo está comendo */
typedef int semaphore;             /* semáforos são um tipo especial de int */
int state[N];                     /* arranjo para controlar o estado de cada um */
semaphore mutex = 1;              /* exclusão mútua para as regiões críticas */
semaphore s[N];                   /* um semáforo por filósofo */

void philosopher(int i)            /* i: o número do filósofo, de 0 a N-1 */
{
    while (TRUE) {                 /* repete para sempre */
        think();                  /* o filósofo está pensando */
        take_forks(i);            /* pega dois garfos ou bloqueia */
        eat();                    /* hummm! Espaguetel */
        put_forks(i);             /* devolve os dois garfos à mesa */
    }
}

void take_forks(int i)              /* i: o número do filósofo, de 0 a N-1 */
{
    down(&mutex);                  /* entra na região crítica */
    state[i] = HUNGRY;             /* registra que o filósofo está faminto */
    test(i);                      /* tenta pegar dois garfos */
    up(&mutex);                   /* sai da região crítica */
    down(&s[i]);                   /* bloqueia se os garfos não foram pegos */
}

void put_forks(i)                  /* i: o número do filósofo, de 0 a N-1 */
{
    down(&mutex);                  /* entra na região crítica */
    state[i] = THINKING;          /* o filósofo acabou de comer */
    test(LEFT);                   /* vê se o vizinho da esquerda pode comer agora */
    test(RIGHT);                  /* vê se o vizinho da direita pode comer agora */
    up(&mutex);                   /* sai da região crítica */
}

void test(i) /* i: o número do filósofo, de 0 a N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

```

O filósofo está pensando e vai comer. A `take_forks` entra na região crítica e fala que ele está faminto. Depois testa se os garfos à direita e à esquerda estão liberados (Os outros filósofos não estão comendo), se estão livres, então o seu estado vai para comendo e sobe mais um no semáforo do filósofo, que sai da região crítica e diminui o semáforo, que não vai ser bloqueado porque foi incrementado mais de uma vez.

Caso um filósofo ao seu lado estivesse comendo, seu estado continuaria no faminto, dessa forma, sairia da região crítica e seu semáforo seria decrementado de forma que ele ficaria bloqueado.

Quando um filósofo que está comendo for devolver os seus garfos, ele vai chamar a test para o filósofo à esquerda e o à direita. Dessa forma, é como se o que estivesse bloqueado tivesse chamado a função test. Então agora o seu estado pode mudar para comendo e o ciclo segue normalmente.

Capítulo 3

3.1 Sem abstração de memória

Cada programa deve referenciar um conjunto privado de endereços de memória que seja local para ele. Caso contrário, uma instrução do tipo JMP pode ir para outro programa, uma vez que são iniciadas de forma sequencial.

Em sistemas embarcados, muitos programas ficam localizados diretamente na ROAM, uma vez que somente os códigos serão conhecidos e o programador tem controle sobre onde estão os dados.

3.2 Abstração de memória: espaços de endereçamento

3.2.1 A noção de espaço de endereçamento

Para dois processos estarem sendo executados simultaneamente, é necessário tomar cuidado com a proteção e realocação.

O espaço de endereçamento cria uma memória abstrata para os processos usarem, onde cada processo tem o seu.

Registradores base e registradores limite

Mapeia cada espaço de endereçamento do processo em uma parte diferente da memória física.

Cada processador tem dois registradores especiais, um base e outro limite. Dessa forma, a base é colocada como a posição inicial do processo na memória e o limite, no final. Quando uma instrução é chamada, é incrementado ao contador base para saber o endereço antes de ir ao barramento de memória. Assim, o código executa JMP 28 e o hardware entende JMP 16412.

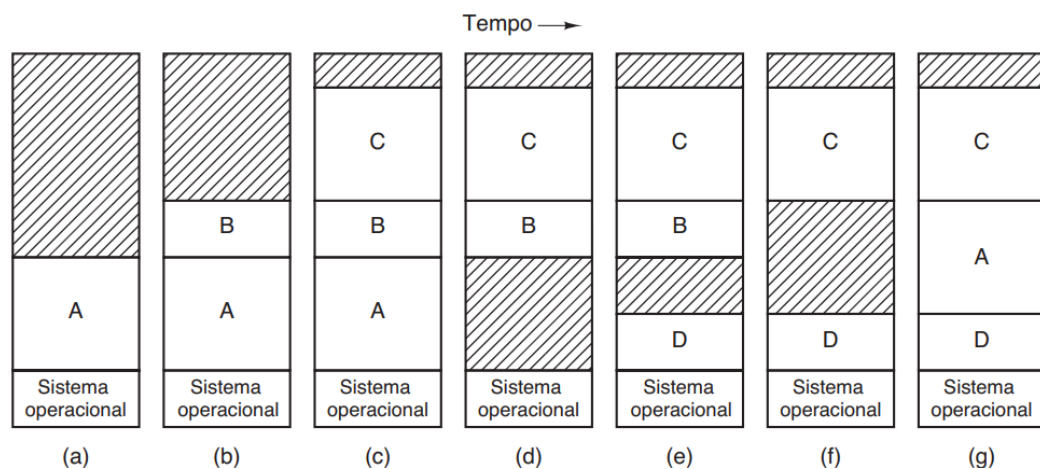
Um problema é que é sempre necessário fazer uma soma e verificar se não ultrapassou o limite.

3.2.1 Troca de memória

Nem todos os processos cabem na RAM.

O swapping traz um processo do disco, coloca na RAM, deixa executar por um tempo e depois retorna ao disco.

Quando um processo entra na memória, ele ocupa um espaço, o próximo que entra ocupa o espaço seguinte e assim por diante. Se um processo fica livre, outro pode entrar no seu lugar, porém podemos ficar com lacunas. Podem ser feitas **compactação de memória** para agrupar todos os programas, mas isso pode demorar bastante.



Quando um processo possui alocação dinâmica e precisa crescer, pode ser que o espaço de memória que está na sua frente está sendo usado por outro processo. Então ele deve ser realocado na memória, realizado o swap, entrar em suspensão até ter uma liberação de memória, ou ser finalizado.

Dessa forma, muitos processos já são inicializados com uma área de heap, que é onde os dados podem crescer sem se preocupar com a compactação,

sendo nele onde alocam-se as variáveis dinâmicas, já as comuns e os endereços de retorno são colocados na pilha

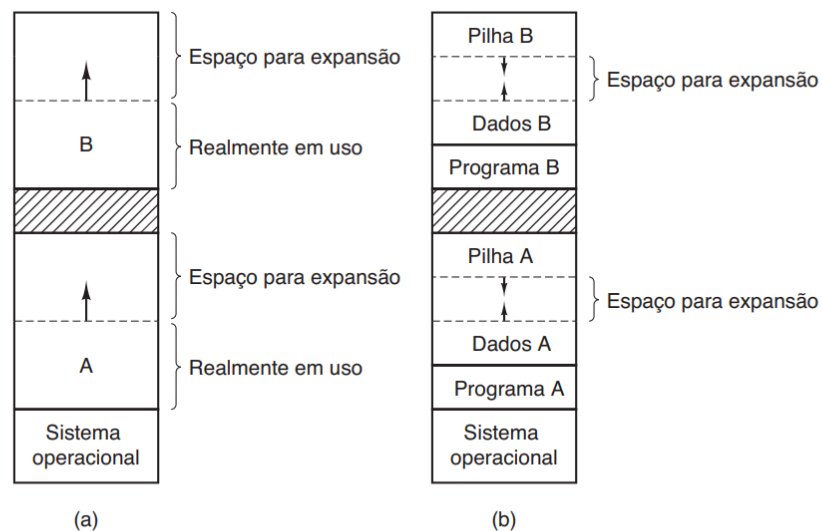


Figura 3.5 (a) Alocação de espaço para um segmento de dados em expansão. (b) Alocação de espaço para uma pilha e um segmento de dados em crescimento.

Na imagem B, caso ocorra um choque entre a Pilha e os Dados, é necessário aumentar o espaço do processo, mudando ele de lugar, colocando em espera, etc.

3.2.1 Gerenciamento de memória livre

Gerenciamento de memória com mapa de bits

A memória é dividida entre unidades de alocação de tamanhos de palavras a kilobytes.

Cada unidade corresponde a um bit no mapa de bits, sendo 0 caso a unidade esteja livre e 1 caso esteja sendo usada.

O tamanho das unidades pode ser escolhido. Caso seja pequeno, precisará de um mapa maior com mais zeros e uns. Porém, caso seja grande, terá um mapa menor, mas pode ocorrer o desperdício de memória. Por exemplo, caso ele tenha tamanho 100, mas está usando somente 5.

Quando um processo vai para a memória e ele utiliza um tamanho K de unidades, é necessário encontrar uma correspondência para esse tamanho onde todos os valores do mapa são 0.

Gerenciamento de memória com lista encadeada

Um segmento de memória pode ser uma área de memória alocada a um processo ou uma área de memória livre entre dois processos.

A lista encadeada terá um valor de P para indicar que é um processo ou L para indicar que é uma região Livre, terá em qual posição da memória começa o processo e qual o seu tamanho, além de um ponteiro para o próximo segmento.

P | 16 | 5 | ponteiro -> Processo começa na posição 16 e possui 5 bytes.

L | 21 | 3 | ponteiro -> Área livre começa na posição 21 e possui 3 bytes.

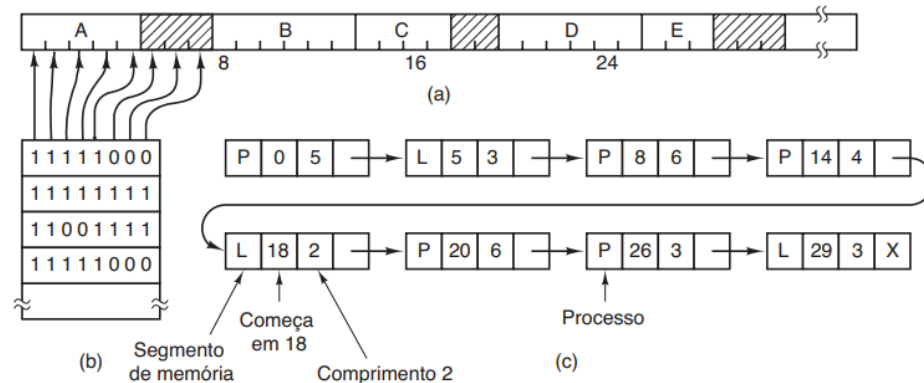


Figura 3.6 (a) Parte da memória com cinco processos e três segmentos de memória. As marcas mostram as unidades de alocação de memória. As regiões sombreadas (0 no mapa de bits) estão livres. (b) O mapa de bits correspondente. (c) A mesma informação como lista.

Se um processo termina e há uma região livre na sua frente ou atrás, a região livre torna-se uma só.

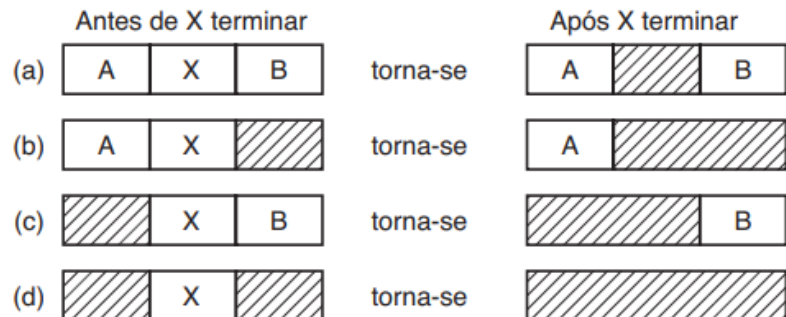


Figura 3.7 Quatro combinações de vizinhos para o processo que termina, X.

Algoritmos para encontrar onde o processo deve ficar:

- **first fit:** Procura desde o início da lista um local que possua espaço suficiente para o processo. Toma o espaço que precisa e transforma o resto em um espaço livre
- **next fit:** Faz uma procura igual ao first fit, mas salva o local onde encontrou espaço. Assim, quando for para a memória de novo, começa a procurar a partir deste local.
- **best fit:** Procura o espaço de tamanho mais próximo ao necessário, evitando modificações nos espaços livres
- **worst fit:** Procura o maior segmento livre possível para deixar um grande espaço para usos futuros de outros processos.

Se fossem feitas listas separadas, uma para processos e outra para segmentos livres, e a segunda fosse ordenada pelo tamanho, o first fit e best fit seriam as melhores opções.

3.3 Memória virtual

Cada processo possui o seu próprio espaço de endereçamento, que é dividido em páginas.

As páginas são mapeadas na memória física, mas não precisam estar lá.

Quando o programa referência uma página que está na memória física, o hardware mapeia dinamicamente. Se não está lá, o sistema operacional vai buscar a página e executa a instrução que falhou pela falta

3.3.1 Paginação

Endereços virtuais são endereços gerados pelo processo.

Em sistemas sem memória virtual, a instrução MOV REG 100, de fato está usando o endereço de memória 100 do hardware e não um endereço virtual.

Quando se utiliza a memória virtual, os acessos aos registradores vão para a MMU, que mapeia um endereço físico a um virtual.

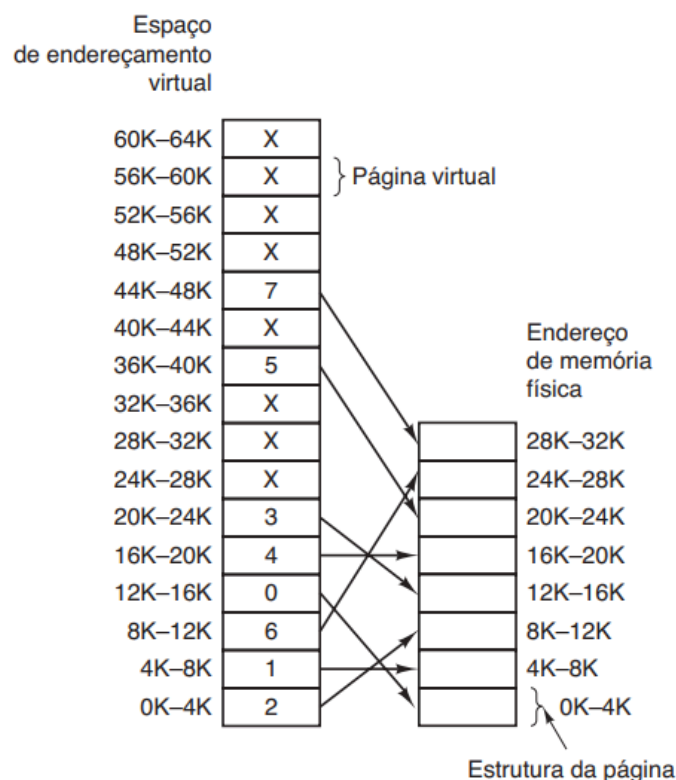


Figura 3.9 A relação entre endereços virtuais e endereços de memória física é dada pela tabela de páginas. Cada página começa com um múltiplo de 4096 e termina 4095 endereços acima; assim, 4K-8K na verdade significa 4096-8191 e 8K-12K significa 8192-12287.

O computador possui 64K de memória virtual e somente 32K de memória física. Cada página, em ambos os lados, possuem tamanho de 4K, dando 16 páginas e 8 molduras.

Supondo que é feito o comando MOV REG, 0. A MMU identifica que o endereço está na página virtual 0, está na moldura de índice 2 (de 8192 a 12287). Então o endereço virtual 0 é transformado para 8192, e o envia para a memória pelo barramento da operação de escrita ou leitura.

A operação MOV REG, 8192 vai para a memória como MOV REG, 24576.

No hardware real, há um bit para indicar se a página está presente ou não na memória física.

Se for solicitado um endereço que está somente na memória virtual, ocorre uma interrupção de page fault, então o sistema operacional usará uma política para remover uma moldura e salvar suas alterações no disco, coloca a página desejada na memória física, atualiza o mapeamento das tabelas e volta para a instrução onde o problema ocorreu.

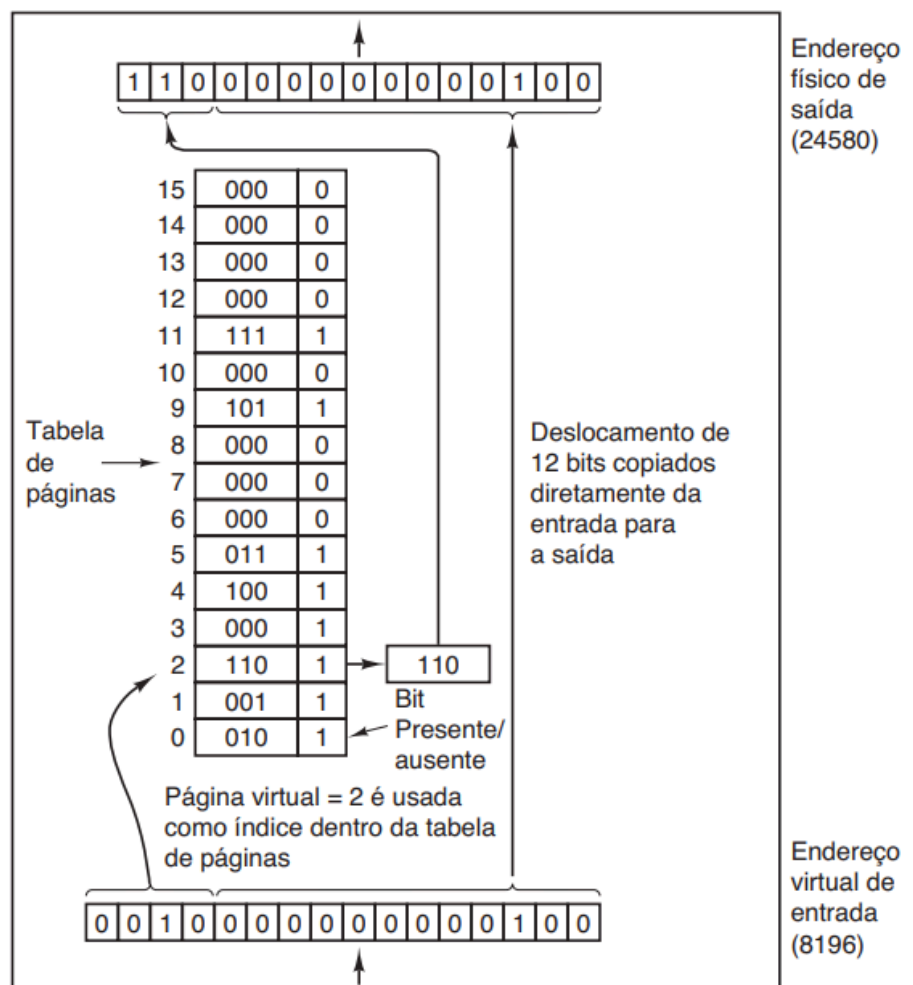


Figura 3.10 Operação interna da MMU com 16 páginas de 4 KB.

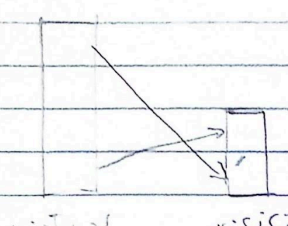
A memória virtual possui 64K (16 bits) e a memória física possui 32K(15 bits)

Cada página foi definida para possuir 4K de endereços (12 bits), dessa forma, como a memória virtual possui 16 bits, os $16-12=4$ bits iniciais são usados para saber de qual página o endereço é.

A tabela de páginas terá justamente 4 bits, ou seja, 16 valores (0 a 15) e ela referenciará uma moldura na memória física. Sabe-se se está referenciado pelo bit Presente/Ausente. Caso ele seja 0 e um dado precisa ser referenciado, então um dos itens da tabela deixará de referenciar uma moldura, que será referenciada por essa página.

Na moldura, os bits que não representam qual a moldura irão receber os valores que não referenciam a página, ou seja, o resto do endereço da memória virtual.

Em resumo, o que está acontecendo é o seguinte: Estamos trocando o cabeçalho da página pelo da moldura, sabendo exatamente qual cabeçalho da moldura está sendo referenciado por qual cabeçalho da página.



virtual física

- A virtual possui mais bits para se endereçar, pois é maior
- A divisão de bits do físico para a virtual determina que tamanho da tabela de mapeamento terá

$$\text{Tam Tabela} = \text{bits Virtual} - \text{bits Físico}$$
- Os bits que armazenam informações na página virtual são todos após os do Tam Tabela, que estão mais à esquerda
- Quando um dado vai ser transferido os bits mais à esquerda são necessários para saber em qual posição da tabela leva o endereço e endereço para a tradução
- Com o bit presente / Ausente igual 0 então ocorre uma interrupção e o SO providencia uma moldura para essa célula da tabela
- Com o Presente / Ausente em 1, podemos saber em qual moldura os dados da página estão
- O valor da moldura para o índice da tabela pode mudar uma vez que, quando necessário substitui uma moldura, o bit presente / Ausente será modificado e outro endereço será enviado para abordar esta página no futuro
- Os dados escritos na moldura não, bits incluindo o índice da moldura concatenado com os outros bits da memória virtual

Jan / Ene	Fev / Feb	Mar / Mar	Abr / Abr	Mai / May	Jun / Jun	Jul / Jul	Ago / Ago	Set / Sep	Out / Oct	Nov / Nov	Dez / Dic
01	02	03	04	05	06	07	08	09	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31					

SP

3.3.2 Tabela de páginas

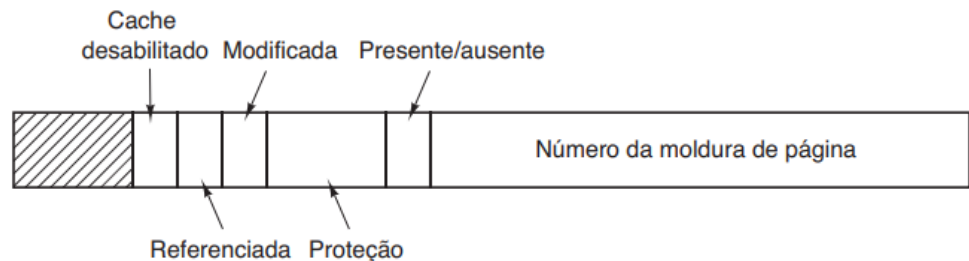


Figura 3.11 Entrada típica de uma tabela de páginas.

O número da moldura de páginas é o valor que está sendo mapeado

O bit presente/Ausente indica se a página está presente na memória ou não.

Proteção indica se pode ser feita leitura/escrita (0), somente leitura(1). Em alguns casos, com 3 bits, indica leitura, escrita e execução.

O bit modificado é colocado como 1 quando há uma modificação na moldura, assim os dados precisam ser modificados na página, caso contrário podem ser descartados.

O bit referenciado indica se a página física está sendo referenciada na tabela.

3.3.3 Acelerando a paginação

O tempo para realizar o mapeamento deve ser mais rápido do que a execução de instruções para evitar gargalos.

Quanto maior o tamanho do processo, maior o tamanho do espaço de endereçamento e, conseqüentemente, maior o tamanho da tabela.

Quando um processo é carregado, todos os seus dados também devem ser carregados. Assim, o custo para recuperar toda a tabela pode ser muito grande.

TLB ou memória associativa

A maioria dos programas fazem um grande número de referências a um mesmo pequeno conjunto de páginas virtuais.

O TLB (translation lookaside buffer) mapeia endereços virtuais na memória física sem precisar de realizar instruções na CPU. Ele diminui a quantidade de itens tendo somente páginas mapeadas de fato na memória real

Sendo que ele fica na MMU e possui um pequeno número de entradas, cada uma com uma verificação se a página é válida (está ou não referenciada), o número da página virtual, bit para identificar modificações na página, código de leitura/escrita/execução e a moldura da página.

Quando um endereço virtual é apresentado para tradução na MMU, verifica-se se o número da página virtual está na TLB, comparando todas as entradas de forma paralela. Se corresponde, então é retornado os dados da TLB. Se o dado quer escrever, mas a permissão é de leitura, é gerada uma protection fault.

Se a página não está presente na TLB (page miss), então a MMU faz uma pesquisa normal na tabela de páginas, trocando uma página da tabela pela que foi procurada.

Gerenciamento da TLB por software

Muitas máquinas fazem esse processamento por software.

As entradas TLB são carregadas pelo SO. Quando há um page miss no TLB, a MMU não busca na tabela de páginas, mas gera uma interrupção e passa o problema pro SO, que encontra a página na tabela ou na memória virtual, troca a mesma na TLB e volta para a instrução interrompida.

Se a TLB for grande o suficiente para reduzir a page miss, pode ser uma boa escolha o seu uso por software, pois diminui o espaço na CPU.

Ausência leve é quando a página não está na TLB, mas está na memória, então não há entrada e saída. Já uma ausência completa é quando a página não está na TLB, não está na memória e deve ser buscada no disco.

3.3.4 Tabelas de páginas para memórias grandes

Tabelas de página multinível

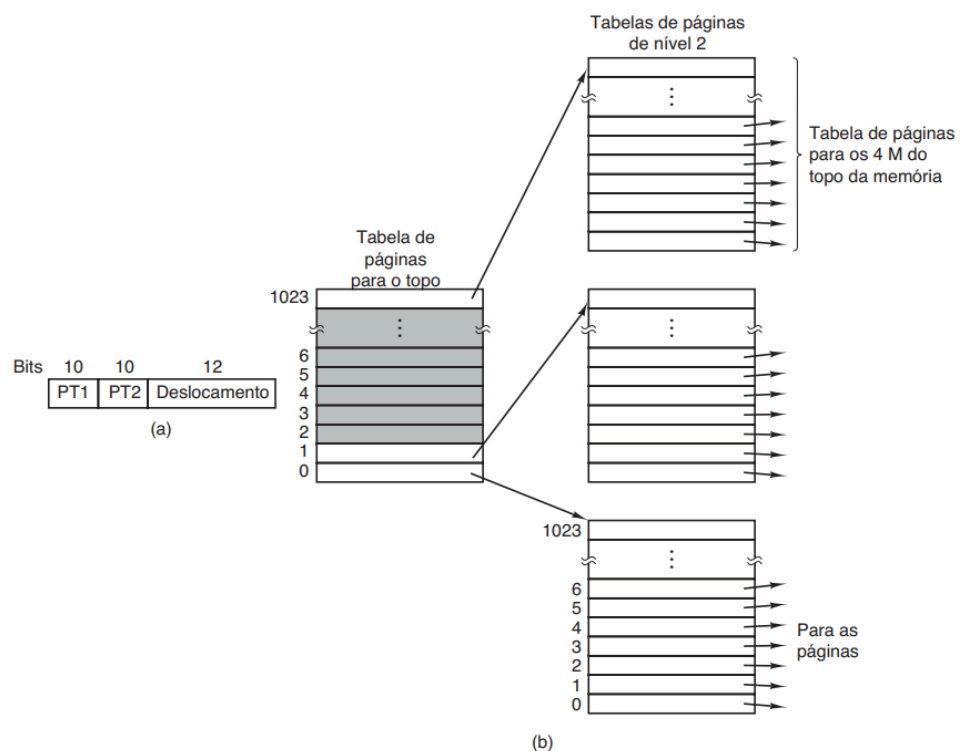


Figura 3.12 (a) Um endereço de 32 bits com dois campos de tabela de páginas. (b) Tabelas de páginas de dois níveis.

Endereço virtual de 32 bits, 10 para PT1, 10 para PT2 e 12 para deslocamento. Logo cada página possui um tamanho de 4K.

A tabela da esquerda possui 1024 entradas, correspondendo ao PT1.

Quando um endereço virtual chega à MMU, o caminho em PT1 é extraído.

Cada linha da tabela é um ponteiro para uma outra tabela, que de fato aponta para as páginas.

Tabelas de páginas invertidas

Existe apenas uma entrada por moldura de página na memória real em vez de uma entrada por página do espaço de endereçamento virtual.

3.4 Algoritmos de substituição de páginas

Não usada recentemente (NRU)

Cada página possui um bit de referenciado (R) e modificado (M).

Quando o processo inicializa, ambos os bits em todas as páginas vão para zero e quando as páginas são referenciadas ou modificadas, os bits vão para um.

Ao ocorrer um page miss, as páginas são divididas em 4 classes:

Classe 0: não referenciada, não modificada.

Classe 1: não referenciada, modificada.

Classe 2: referenciada, não modificada.

Classe 3: referenciada, modificada.

O algoritmo remove uma página aleatória de ordem mais baixa e que não está vazia.

A cada tique do relógio, o valor de R é limpo e a página que está sendo usada coloca ele como 1. Assim, sabemos que a página está sendo utilizada e vale mais a pena remover uma modificada do que uma referenciada que está sendo usada.

Primeiro a entrar, primeiro a sair (FIFO)

Lista encadeada onde o primeiro item é o mais antigo e o a ser removido.

Algoritmo de substituição de página segunda chance

Melhora o FIFO

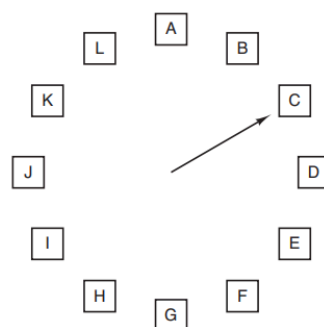
Se o bit R for 0, remove a página. Se o bit R for 1, coloca ele como zero e envia para o final da lista e continua a procurar uma página que não foi referenciada no último intervalo do relógio.

Se todas as páginas já possuem o bit R em 1, então o FIFO normal é usado.

Substituição de página do relógio

Mantém todos os itens em uma lista circular.

Se o bit R for 0, a página é removida e a nova página é inserida no lugar, o ponteiro avança uma posição. Se R for 1, ele é zerado e vai para a próxima página, repetindo até encontrar uma página com R = 0



Quando ocorre uma falta de página, a página indicada pelo ponteiro é inspecionada. A ação executada depende do bit R:

R = 0: Remover a página

R = 1: Zerar R e avançar o ponteiro

|| **Figura 3.15** Algoritmo de substituição de página relógio.

Usada menos recentemente (LRU)

Remove a página que foi menos utilizada no intervalo de tempo mais longo.

Possui um contador que é incrementado a cada instrução.

A cada referência à memória, o valor do contador é armazenado na entrada da tabela de páginas para a página que acabou de ser referenciada.

A página que possui o menor contador é removida

Simulação do LRU em software (aging)

Aplica o NFU (Not frequently used)

Possui contadores em software, cada um relacionado a uma página.

A cada clock, o valor do bit R é adicionado à esquerda do contador contador e o seu valor é decrementado ou incrementando, dependendo do valor de R

A página que possuir a menor contagem é removida.

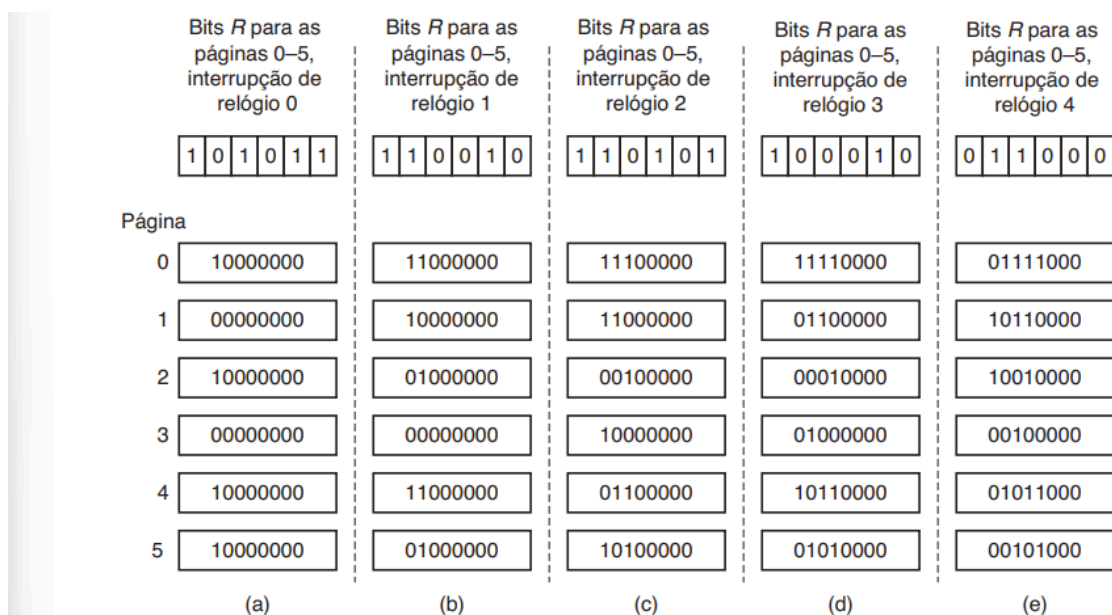


Figura 3.17 O algoritmo de envelhecimento simula o LRU em software. São mostradas seis páginas para cinco interrupções de relógio. As cinco interrupções de relógio são representadas de (a) até (e).

3.7 Segmentação

Para alguns problemas, ter mais de um espaço de endereçamento de memória pode ser mais vantajoso.

A máquina possui diferentes espaços de endereçamento independentes chamados de **segmentos**. Cada segmento vai de 0 até um valor máximo.

Os segmentos podem possuir tamanhos diferentes e que mudam de tamanho conforme a execução do programa.

O programador sabe que está utilizando segmentos e pode gerar rotinas para eles, pois sabe o que cada segmento possui.

A segmentação pode ser entendida como uma DLL, um conjunto de código que já foi compilado e já possui o seu espaço de endereçamento existente e conhecido.

Capítulo 4

Arquivos são espaços de endereçamento para modelar o disco

4.1.2 Estrutura de arquivos

Os arquivos podem ser estruturados de três formas principais:

A primeira forma é uma sequência desestruturada de bytes, com significados impostos por programas de usuário.

Na segunda forma, estruturando os bytes, cada registro possui uma certa quantidade específica de tamanho de dados. Uma leitura retorna uma sequência e uma escrita sobrepõe ou anexa um registro.

A terceira forma é criar uma árvore, cada uma com um campo-chave em uma posição do registro. Dessa forma os arquivos ficam subdivididos na árvore.

4.1.3 Tipos de arquivos

Arquivos regulares possuem informações do usuário, em geral escritos com caracteres do tipo ASCII ou binários, com uma estrutura interna conhecida.

Diretórios são arquivos do sistema que mantêm a estrutura do sistema de arquivos.

4.3 Implementação do sistema de arquivos

4.3.2 implementação de arquivos

Alocação contígua

É simples de implementar, pois basta saber o endereço do primeiro bloco e a quantidade de blocos, já que todos possuem o mesmo tamanho.

Os arquivos podem ser lidos com apenas uma operação do disco.

Porém com o tempo o disco fica fragmentado pois arquivos saíram e deixaram espaços em branco.

Alocação por lista encadeada

A primeira palavra de cada bloco é usada como ponteiro para o próximo e o restante do bloco é usado para dados.

Nenhum espaço é perdido pela fragmentação. É necessário ter em disco o endereço somente do primeiro bloco.

Para chegar ao bloco n , é necessário percorrer $n-1$ blocos.

Alocação por lista encadeada usando uma tabela na memória

Cada bloco é colocado em uma posição de uma tabela, sendo que o último bloco tem valor de $n-1$. O que indica qual a posição do próximo item é um inteiro na tabela. A tabela é chamada de FAT, que precisa estar toda carregada na memória.

I-nodes

É uma tabela que contém os índices dos blocos e onde eles estão no disco. O último dado da tabela é um ponteiro para uma outra tabela de dados, uma vez que a primeira pode estar cheia.

Hard links

No disco, os arquivos são referenciados pelo seu i-node. O hard link faz com que o i-node de um arquivo seja o do outro. Dessa forma, é como se ambos

apontassem para o mesmo endereço de memória. Por isso só funciona no mesmo sistema de arquivos

Se o arquivo original for deletado, ainda é possível acessar os dados, pois a referência da memória ainda não foi perdida

Soft links

São atalhos de um arquivo para outro e funcionam em sistemas de arquivos diferentes.

Se colocar para ler o arquivo que é um soft link, estará lendo na verdade os dados do arquivo referenciado.

Caso o arquivo referenciado seja deletado, o soft link para de funcionar

Capítulo 5

Dispositivos de blocos: Possuem blocos para armazenar os dados, como o HDD

Dispositivos de caracteres: Envia ou recebe um fluxo de caracteres sem considerar qualquer estrutura de blocos. Teclado, mouse, impressora.

Disco de HDD: Contém um preâmbulo (Endereço, tamanho do setor e dados similares), depois os bits de dados e, por fim, uma soma de correção de erro (ECC)

DMA (Acesso direto à memória)

A CPU programa um controlador DMA para gerenciar transferências de dados, permitindo mover dados entre dispositivos de E/S sem passar pela CPU

Quando inicializa a transferência, o controlador assume o barramento e quando termina, avisa a CPU por meio de uma interrupção.

Discos

Preâmbulo: Inicia com um padrão binário para permitir que o hardware identifique o começo do setor.

Campo de dados: Em geral, possui 512 bytes.

ECC: informações redundantes para recuperação de erros de leitura.

O setor 0 de cada trilha começa sempre um pouco antes ou depois da trilha anterior para que, quando seja preciso mudar de trilha, não se perca a posição que estava, tendo que esperar uma nova rotação completa.

Algoritmos de escalonamento de braços e disco

O tempo necessário para ler ou escrever um dado no disco depende do tempo de rotação, tempo de posicionamento do braço e o tempo real de transferência do dado.

Primeiro a chegar, primeiro a ser servido: Atende as requisições na ordem em que chegam.

Posicionamento mais curto primeiro (SSF): O que está mais próximo será atendido primeiro.

Algoritmo do elevador: Realiza todas as requisições que estão apontadas para uma direção e, quando todas finalizarem, então muda a direção. Feita a partir de um bit SOBE ou DESCE os cilindros.

Hoje em dia, vários setores são lidos e armazenados em cache