CAPÍTULO IV

ALGORITMOS DE MUDANÇA DE PÁGINA

No momento em que é detectada a falta de uma página (i.e., ocorre um ***page fault***), o S.O. deve escolher uma das páginas atualmente residentes na memória para ser removida, de forma a liberar um quadro de página para a colocação da página faltante. Deve-se tomar o cuidado, caso a página a ser removida tenha sido alterada, de reescrevê-la no disco.

O problema que surge aqui é o seguinte: de que forma escolher a página a remover ? É claro que devemos procurar remover uma página que não seja muito necessária aos processos em execução, isto é, devemos remover uma página não muito utilizada. Se, ao contrário, removêssemos uma página amplamente utilizada, em pouco tempo uma referência à mesma seria realizada, o que ocasionaria um page-fault e a necessidade de recarregar esta página, além de escolher outra para ser removida.

O que devemos considerar então é o seguinte: como determinar qual página não está sendo muito utilizada? Os próximos ítens discutiram alguns métodos de se realizar essa escolha.

1. Mudança Ótima de Página

O método ótimo de mudança de página consiste no seguinte:

- determinar, para cada página presente na memória, quantas instruções faltam para a mesma ser referenciada (i.e., quantas instruções serão executadas antes que a página seja referenciada);
- retirar a página que for demorar mais tempo para ser novamente referenciada.

Como é fácil perceber, o método delineado acima não pode ser implementado numa situação real, pois exigiria conhecimento de situações futuras (de que forma um S.O. conseguiria determinar quantas instruções faltam para uma página ser referenciada?). Este método é utilizado apenas em simulações, onde temos um controle completo das execuções dos processos, para fins de comparação de algum algoritmo com o algoritmo ótimo.

2. Mudança da Página não Recentemente Usada

Este método seleciona para retirar uma página que não tenha sido recentemente utilizada. Para determinar se uma página foi ou não utilizada recentemente, conta-se com o auxílio de 2 bits associados com cada uma das páginas na memória:

R: indica se a página já foi referenciada;

M: indica se a página foi modificada.

Estes bits, presentes em muitos hardwares de paginação, são alterados automaticamente (por hardware) quando uma referência à página é realizada. Assim, quando uma instrução lê um dado em uma certa posição de memória, o bit R da página correspondente é automaticamente colocado em 1; quando uma instrução escreve em uma dada posição de memória, os bits **R** e **M** da página correspondente são colocados em 1. Esses bits são colocados inicialmente em 0 quando uma página é carregada na

memória. Uma vez que esses bits sejam colocados em 1, eles só podem voltar a 0 através de instruções em software.
O método de determinar se uma página foi recentemente utilizada é então o seguinte:

- a cada tiquetaque do relógio, o S.O. coloca todos os bits R das páginas em 0;
- quando ocorre um ***page fault***, temos três categorias de páginas:
  classe 0: R=0, M=0
  classe 1: R=0, M=1
  classe 2: R=1, M=0
  classe 3: R=1, M=1

A classe 2 ocorre para uma página que foi referenciada pelo menos uma vez desde o último tiquetaque do relógio, sem ter sido alterada. A classe 3 ocorre para uma página que já foi alterada e referenciada desde o último tiquetaque. As classes 0 e 1 são as correspondentes às classes 2 e 3 (respectivamente), que não tiveram nenhuma referência desde o último tiquetaque do relógio.

- escolhe-se para ser retirada uma das páginas que pertencer à classe mais baixa não vazia, no momento da ocorrência do ***page fault***.

Esse algoritmo apresenta as vantagens de ser implementável de forma simples e eficiente.
Quando o hardware não dispõe dos bits **R** e **M**, o S.O. pode simular esses bits através da utilização dos mecanismos de proteção de páginas, da seguinte forma:

- Inicialmente, marcam-se todas as páginas como ausentes;
- quando uma página é referenciada, é gerado um page fault (pois a página está marcada como ausente);
- o S.O. então marca essa página como presente, mas permite apenas leitura (i.e., Read-Only);
- o S.O. marca também, em uma tabela interna, um bit R simulado para essa página;
- quando uma escrita for tentada nessa página, uma interrupção de acesso inválido será gerada, e o S.O. poderá então marcar a página como de leitura e escrita (Read-Write);
- o S.O. então marca numa tabela interna o bit M correspondente a essa página.

3. Mudança de Página "Primeira a Entrar, Primeira a Sair"(FIFO)

Neste caso, mantém-se uma fila de páginas referenciadas. Ao entrar uma nova página, ela entra no fim da fila, substituindo a que estava colocada no início da fila. O problema com este algoritmo é que pode retirar páginas muito valiosas (i.e., páginas que, apesar de estarem a bastante tempo na memória estão sendo amplamente utilizadas).
Uma possível solução para esse problema consiste na utilização dos bits **R** e **M**:

- verifica os bits R e M da página mais antiga;
- se essa página for classe 0, escolhe-a para ser retirada;
- se não for, continua procurando na fila;
- se em toda a fila não achou nenhum classe 0, prossegue para as classes seguintes (1, 2 e 3).

Uma variação dessa solução é o algoritmo conhecido como segunda chance:

- verifica o bit R da página mais velha;
- se R for zero, utiliza essa página;

- se R for 1, põe R em 0 e coloca a página no fim da fila;
- prossegue analisando a fila, até encontrar uma página com R=0 (no pior caso, será a primeira página, que teve seu R alterado para 0).

4. Mudança da Página Menos Recentemente Utilizada (Least Recently Used, LRU).

Este método se baseia nas seguintes observações:

- páginas muito utilizadas nas instruções mais recentes provavelmente permanecerão muito utilizadas nas próximas instruções;
- páginas que não são utilizadas a tempo provavelmente não serão utilizadas por bastante tempo.

O algoritmo então consiste no seguinte: quando ocorre um ***page fault***, retira a página que a mais tempo não é referenciada. O problema básico com este algoritmo é que a sua implementação é muito dispendiosa: deve-se manter uma lista de todas as páginas na memória, com as mais recentemente utilizadas no começo e as menos recentemente utilizadas no final; e esta lista deve ser alterada a cada referência de memória. Note-se que a implementação por software é completamente inviável em termos de tempo de execução.

As duas possíveis soluções por hardware são:

Na primeira, mantemos no processador um contador **C**, com um número relativamente grande de bits (p.ex.: 64 bits). Com esse contador, realizamos o seguinte procedimento:

- cada instrução executada, incrementamos esse contador; cada entrada na tabela de páginas, tem também associado um campo com tantos bits quantos os do contador **C**; depois de cada referência à memória, o contador **C** é armazenado na entrada correspondente à página referenciada; quando ocorre um **page fault**, escolhemos a página com o menor valor no campo de armazenamento do contador (pois essa será a página que foi referenciada a mais tempo).

Na segunda solução, para ***n*** quadros de páginas, temos uma matriz ***n*** X ***n*** de bits, e agimos da seguinte forma:

- quando a página i é referenciada, colocamos em 1 todos os bits da linha i da tabela e, em seguida, colocamos em 0 todos os bits da coluna i da tabela;
- quando ocorre um page fault, retiramos a página que apresentar o menor número em sua linha (este número é interpretado pela associação de todos os bits da linha da esquerda para a direita).

5. Simulando LRU em Software

Nas soluções de LRU apresentadas até agora, consideramos sempre a necessidade de hardware especial. Mas, e quando não existe hardware especial para LRU?
Com essa consideração em vista, desenvolveu-se uma aproximação para LRU, chamada

NFU (não freqüentemente utilizada), que pode ser realizada por software, da seguinte forma:

- um contador é mantido, em software, para cada página;
- a cada interrupção de relógio, o bit R correspondente a cada uma das páginas é somado a esse contador (portanto, se a página foi referenciada dentro desse intervalo de relógio, o valor do contador será incrementado, caso contrário, o valor do contador será mantido);
- quando ocorre um page fault, escolhe-se a página com valor menor nesse contador.

O grande problema com esse algoritmo é que, se uma página intensivamente utilizada durante um certo tempo, ela tender a permanecer na memória, mesmo quando não for mais necessária (pois adquiriu um valor alto no contador).

Felizmente, uma modificação simples pode ser feita no método evitando esse inconveniente. A alteração consiste no seguinte:

- os contadores são deslocados 1 bit à direita, antes de somar **R**;
- **R** é somado ao bit mais à esquerda (mais significativo) do contador, ao invés de ao bit mais à direita (menos significativo).

Esse algoritmo é conhecido como algoritmo de ***aging***. É obvio que uma página que não foi referenciada a 4 tique-taques ter 4 zeros à esquerda e, portanto um valor baixo, colocando-a como candidata à substituição.

Este algoritmo tem duas diferenças fundamentais em relação ao algoritmo de LRU:

1. não consegue decidir qual a referencia mais recente com intervalos menores que um tiquetaque;
2. o número de bits do contador é finito e, portanto, quando este chega a zero, não conseguimos mais distinguir as páginas mais antigas.

Por exemplo, com 8 bits no contador, uma página não referenciada a 9 tique-taques não pode ser distinguida de uma página não referenciada a 1000 tique-taques. No entanto, em geral 8 bits é suficiente para determinar que uma página já não é mais necessária.

6. Considerações de Projeto para Sistemas de Paginação

Veremos alguns pontos que devem ser considerados, além do algoritmo de paginação propriamente dito, para que um sistema de paginação tenha um bom desempenho.

6.1. Modelo do Conjunto Ativo (*Working Set*)

Num sistema puro de paginação, que também pode ser chamado de paginação por demanda, o sistema começa sem nenhuma página na memória e elas vão sendo carregadas à medida que forem necessárias, seguindo o algoritmo de substituição de página escolhido.

Esta estratégia pura pode ser melhorada, como veremos a seguir.

Um primeiro dado importante a considerar é a existência, na grande maioria dos processos, de uma localidade de referências, o que significa que os processos mantêm, em cada uma das fases de sua execução, referências a frações pequenas do total do

número de páginas necessárias a ele. Um exemplo disso é um compilador, que durante um certo tempo acessa as páginas correspondentes à análise sintática, depois essas páginas deixam de ser necessárias, passando-se então a utilizar as de análise semântica e assim por diante.
Baseado neste fato, surge o conceito de conjunto ativo (***working set***), que corresponde ao conjunto de páginas correntemente em uso de um dado processo. É fácil de notar que, se todo o conjunto ativo de um processo estiver na memória principal, ele executará praticamente sem gerar ***page faults***, até que passe a uma nova fase do processamento. Por outro lado, se não houver espaço para todo o conjunto ativo de um processo, este gerará muitos ***page faults***, o que ocasionará uma grande diminuição em seu tempo de execução, devido à necessidade de constantes trocas de páginas entre memória e disco.
Aqui surge o conceito de ***thrashing***, que ocorre quando um processo gera muitos ***page fault*** em poucas instruções.
Surge então a seguinte consideração: o que fazer quando um processo novo é iniciado ou, num sistema com swap, quando um processo que estava em disco deve ser colocado na memória? Se deixarmos que esse processo gere tantos ***page fault*** quanto necessários para a carga do seu ***working set***, teremos uma execução muito lenta. Devemos então, de alguma forma, determinar qual é o ***working set*** do processo e carregá-lo na memória antes de permitir que o processo comece a executar. Este é o chamado modelo do conjunto ativo (***working set model***). O ato de carregamento adiantado das páginas (antes da ocorrência do ***page fault*** para a mesma) é chamado pré-paginação.
Vejamos agora algumas considerações com relação aos tamanhos dos ***working set***. Se a soma total dos ***working set*** de todos os processos residentes em memória é maior que a quantidade de memória disponível, então ocorrer ***thrashing***. (Obs: processos residentes na memória são aqueles que o escalonador de baixo nível utiliza para a seleção do atualmente em execução. O escalonador em alto nível é o responsável pela troca (***swap***) dos processos residentes em memória, a certos intervalos de tempo.)
Portanto, devemos escolher os processos residentes em memória de forma que a soma de seus ***working set*** não seja maior que a quantidade de memória disponível.
Após os fatos apresentados acima, surge a questão: como determinar quais as páginas de um processo que fazem parte de seu ***working set***? Uma possível solução para isto é utilizar o algoritmo de ***aging***, considerando como parte do ***working set*** apenas as páginas que apresentem ao menos um bit em 1 em seus primeiros n bits (i.e., qualquer página que não seja referenciada por n tique-taques consecutivos é retirada do ***working set*** do processo). O valor de n deve ser definido experimentalmente.

6.2. Rotinas de Alocação Local X Global

Devemos agora determinar de que forma a memória será alocada entre os diversos processos executáveis que competem por ela.
As duas estratégias básicas são chamadas de alocação local e alocação global, que são descritas a seguir:

- alocação local: quando é gerado um ***page fault*** em um dado processo, retiramos uma das páginas do próprio processo, para a colocação da página requerida;
- alocação global: quando um ***page fault*** é gerado, escolhe para retirar uma das páginas da memória, não importa a qual processo ela esteja alocada.

Para exemplificar a diferença entre as duas técnicas, veja a Figura 1 onde, na situação da figura (a), é gerado um ***page fault*** para a página A6 (os números ao lado das páginas indicam o valor do contador do algoritmo de ***aging***). Se seguimos uma estratégia local, a página retirada será a A5 (conforme (b)), enquanto que para uma estratégia global, a página retirada será a B3.

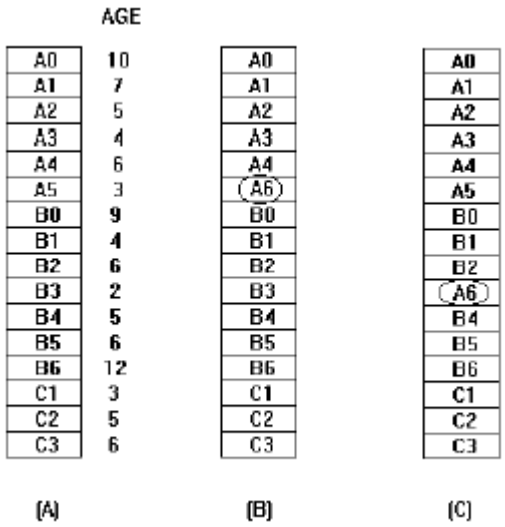

Figura 1. Exemplo das duas técnicas de alocação local e global em ação.

É fácil de notar que a estratégia local mantém fixa a quantidade de memória destinada a cada processo, enquanto que a estratégia global muda dinamicamente a alocação de memória.
Em geral, os algoritmos globais apresentam maior eficiência, devido aos seguintes fatos:

- se o ***working set*** de um processo aumenta, a estratégia local tende a gerar ***thrashing***;
- se o ***working set*** de um processo diminui, a estratégia local tende a desperdiçar memória.

Entretanto, se nos decidimos por utilização de uma estratégia global, devemos resolver a seguinte questão: de que forma o sistema pode decidir dinamicamente (i.e., a todo instante) quantos quadros de página alocar para cada processo (em outras palavras: qual o tamanho do ***working set*** do processo em cada instante?).
Uma possível solução é realizar a monitoração pelos bits de ***aging***, conforme indicado acima. Entretanto isto apresenta um problema, o ***working set*** de um processo pode mudar em poucos microssegundos, enquanto que as indicações de aging só mudam a cada tique-taque do relógio (digamos, 20ms).
Uma solução melhor para esse problema é a utilização de um algoritmo de alocação por freqüência de falta de página (PFF: ***page fault frequency***). Esta técnica é baseada no fato de que a taxa de falta de páginas para um dado processo decresce com o aumento do número de quadros de página alocados para esse processo.Ver a Figura 2.

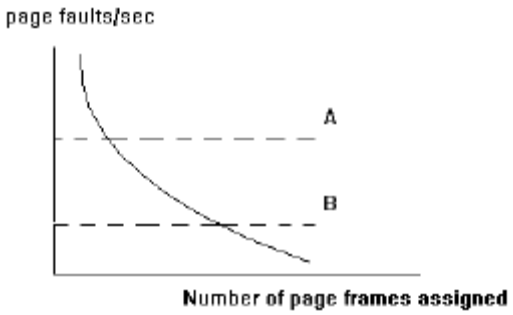

Figura 2. Relação entre o número de quadro de páginas e a taxa de falta de páginas

6.3. Tamanho da Página

Um outro fator que deve ser cuidadosamente estudado é o tamanho da página. Existem considerações em favor tanto de páginas pequenas quanto de páginas grandes, como são:

Considerações a favor de páginas pequenas:

- na média, metade da página final de qualquer segmento é desperdiçada (pois os programas e áreas de dados não terminam necessariamente em múltiplos de página). Isto é conhecido como fragmentação interna;
- um programa que consista em processos pequenos pode executar utilizando menos memória quando o tamanho da página é pequeno.

Considerações a favor de páginas grandes:

- quanto menor o tamanho da página, maior o tamanho da tabela de páginas (para uma dada quantidade de memória virtual);
- quando uma página precisa ser carregada na memória, temos um acesso ao disco, com os correspondentes tempos de busca e espera de setor, além do de transferência. Assim, se já transferimos uma quantidade maior de bytes a cada acesso, diminuímos a influência dos dois primeiros fatores do tempo, aumentando a eficiência;
- se o processador central precisa alterar registradores internos referente à tabela de páginas a cada chaveamento de processo, então páginas maiores necessitarão menos registradores, o que significa menor tempo de chaveamento de processos.

Tamanhos comumente encontrados de página são: 512 bytes, 1 kbytes, 2 kbytes e 4 kbytes.

6.4. Considerações de Implementação

Normalmente, quando queremos implementar um sistema de paginação, devemos considerar alguns fatos que complicam um pouco seu gerenciamento, apresentaremos alguns exemplos a seguir.

i) **Bloqueamento de Páginas na Memória**

Quando um processo bloqueia aguardando a transferência de dados de um dispositivo para um buffer, um outro processo é selecionado para execução. Se este outro processo gera um ***page fault***, existe a possibilidade de que a página escolhida para ser retirada seja justamente aquela onde se encontra o buffer que estava aguardando a transferência do

dispositivo. Se o dispositivo tentar então transferir para esse buffer, ele pode estragar a nova informação carregada pelo page fault. Existem basicamente duas soluções para este problema:

1. trancamento da página que contém o buffer, impedindo que a mesma seja retirada da memória;
2. realizar as transferências de dispositivos sempre para buffers internos do kernel e, após terminada a transferência, destes para o buffer do processo.

ii) **Páginas Compartilhadas**

Um outro problema que surge está relacionado com o fato de que diversos processos podem estar, em um dado momento, utilizando o mesmo programa (p.ex.: diversos usuários rodando um editor, ou um compilador). Para evitar duplicação de informações na memória, é conveniente que essas informações comuns sejam armazenadas em páginas que são compartilhadas pelos processos que as utilizam.

Um problema que surge logo de início é o seguinte: algumas partes não podem ser compartilhadas como, por exemplo, os textos que estão sendo editados em um mesmo editor. A solução para este problema é simples: basta saber quais as páginas que são apenas de leitura (p.ex.: o código do programa), permitindo que estas sejam compartilhadas e impedindo o compartilhamento das páginas que são de leitura e escrita (p.ex.: o texto sob edição).

Um outro problema mais difícil de resolver é o seguinte:

- suponha que dois processos, A e B, estão compartilhando o código de um editor;
- se A termina a edição e, portanto, sai da memória algumas de suas páginas (as que correspondem ao código do editor) não poderão ser retiradas devido a serem compartilhadas, pois se isto ocorresse, seriam gerados muitos ***page fault*** para o processo B, o que representaria muita sobrecarga para o sistema;
- se um dos processo é enviado para o disco (swap) temos uma situação semelhante ao caso do término de um deles (como acima).

A solução para isto é manter alguma forma de estrutura de dados que indique quais das páginas residentes estão sendo compartilhadas. Este processo, entretanto, não é simples.