Capítulo 17

Paginação em disco

A memória RAM sempre foi um recurso escasso em sistemas de computação. Por isso, seu uso deve ser gerenciado de forma eficiente, para que todos os processos possam ter memória suficiente para operar. Além disso, a crescente manipulação de informações multimídia (imagens, áudio, vídeo) contribui para esse problema, uma vez que essas informações são geralmente volumosas e seu tratamento exige grandes quantidades de memória livre.

Como a memória RAM é um recurso caro (cerca de U$10/GByte no mercado americano, em 2018) e que consome uma quantidade significativa de energia, aumentar a quantidade de memória nem sempre é uma opção factível. No entanto, o computador geralmente possui discos rígidos ou SSD maiores, mais baratos e mais lentos que a memória RAM. Em valores de 2018, 1 GByte de disco rígido custa cerca de U$0,05, enquanto 1 GByte de SSD custa cerca de U$0,30 (valores apenas indicativos, variando de acordo com o fabricante e a tecnologia envolvida).

Este capítulo apresenta técnicas usadas para usar um dispositivo de armazenamento secundário como extensão da memória RAM, de forma transparente para as aplicações.

17.1 Estendendo a memória RAM

Os mecanismos de memória virtual suportados pelo hardware, apresentados no Capítulo 15, permitem usar dispositivos de armazenamento secundário como extensão da memória RAM. Com isso, partes ociosas da memória podem ser transferidas para um disco, liberando a memória RAM para outros usos. Caso algum processo tente acessar esse conteúdo posteriormente, ele deverá ser trazido de volta à memória. A transferência dos dados entre memória e disco é feita pelo sistema operacional, de forma transparente para os processos.

Existem diversas técnicas para usar um espaço de armazenamento secundário como extensão da memória RAM, com ou sem o auxílio do hardware. As mais conhecidas são:

Overlays: o programador organiza seu programa em módulos que serão carregados em uma mesma região de memória em momentos distintos. Esses módulos, chamados de *overlays*, são gerenciados através de uma biblioteca específica. Por exemplo, o código de um compilador pode separar o analisador léxico, o analisador sintático e o gerador de código em *overlays*, que serão ativados

em momentos distintos. Esta abordagem, popular em alguns ambientes de desenvolvimento nos anos 1970-90, como o Turbo Pascal da empresa Borland, é raramente usada hoje em dia, pois exige maior esforço por parte do programador.

Swapping: consiste em mover um processo ocioso da memória RAM para um disco (*swap-out*), liberando a memória para outros processos. Mais tarde, quando esse processo for acordado (entrar na fila de prontos do escalonador), ele é carregado de volta na memória (*swap-in*). A técnica de *swapping* foi muito usada até os anos 1990, mas hoje é pouco empregada em sistemas operacionais de uso geral.

Paging: consiste em mover páginas individuais, conjuntos de páginas ou mesmo segmentos da memória para o disco (*page-out*). Se o processo tentar acessar uma dessas páginas mais tarde, a MMU gera uma interrupção de falta de página e o núcleo do SO recarrega a página faltante na memória (*page-in*). Esta é a técnica mais usada nos sistemas operacionais atuais, por sua flexibilidade, rapidez e eficiência.

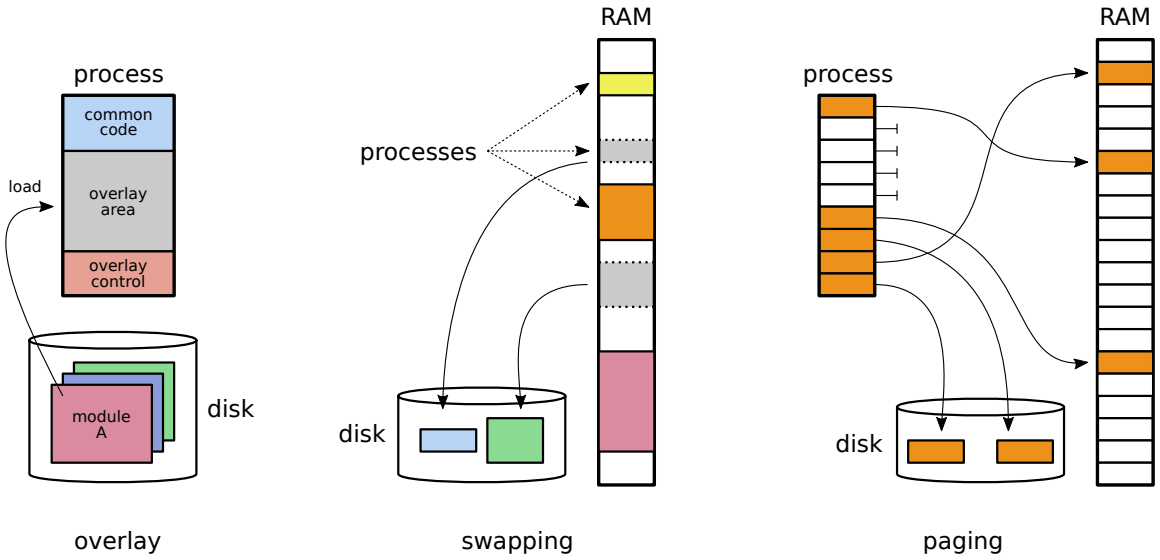

Figura 17.1: Abordagens de extensão da memória em disco.

17.2 A paginação em disco

A ideia central da paginação em disco consiste em transferir páginas ociosas da memória RAM para uma área em disco, liberando memória para outras páginas. Esta seção explica o funcionamento básico desse mecanismo e discute sobre sua eficiência.

17.2.1 Mecanismo básico

A transferência de páginas entre a memória e o disco é realizada pelo núcleo do sistema operacional. As páginas a retirar da memória são escolhidas por ele, de acordo com algoritmos de substituição de páginas, discutidos na Seção 17.3. Quando um processo tentar acessar uma página que está em disco, o núcleo é alertado pela MMU e traz a página de volta à memória para poder ser acessada.

Para cada página transferida para o disco, a tabela de páginas do processo é ajustada: o *flag* de presença da página em RAM é desligado e a posição da página no disco é registrada, ao invés do quadro. Essa situação está ilustrada de forma simplificada na Figura 17.2.

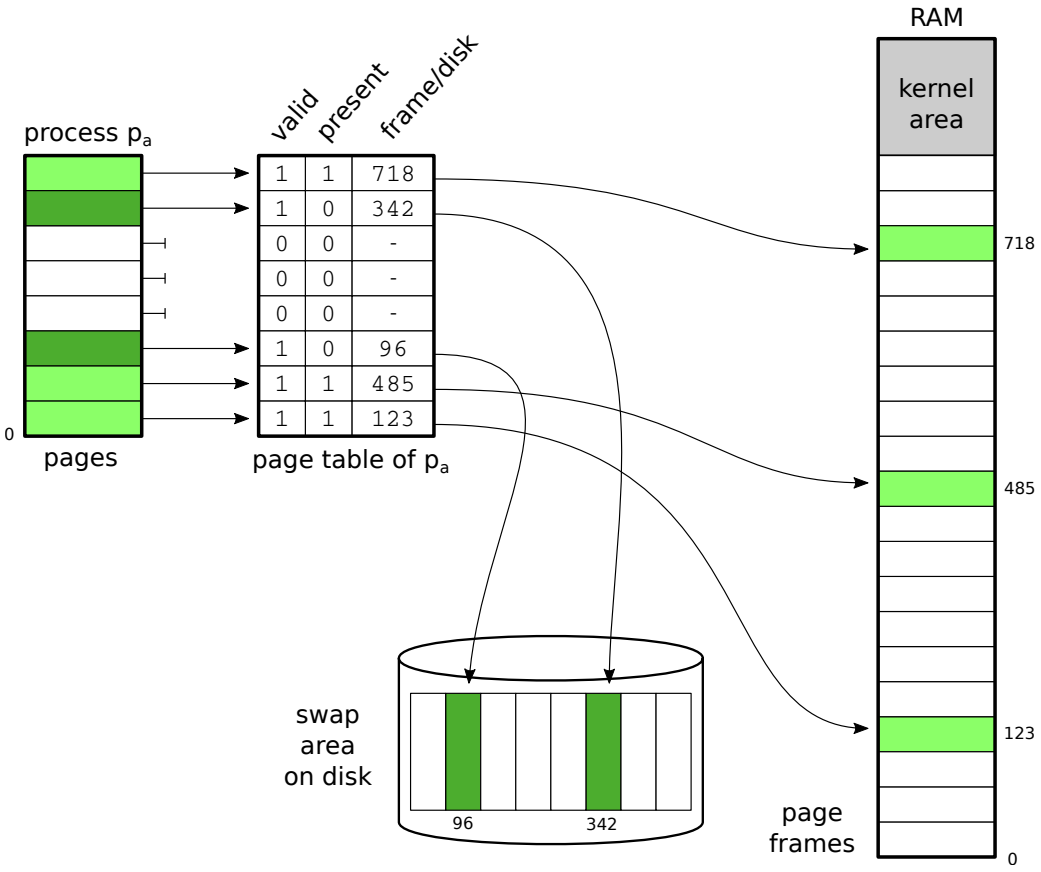

Figura 17.2: A paginação em disco.

O armazenamento externo das páginas pode ser feito em um disco exclusivo (usual em servidores de maior porte), em uma partição do disco principal (usual no Linux e outros UNIX) ou em um arquivo reservado dentro do sistema de arquivos (como no Windows NT e sucessores). Em alguns sistemas, é possível usar uma área de troca remota, em um servidor de rede; todavia, essa solução apresenta baixo desempenho. Por razões históricas, essa área de disco é geralmente denominada *área de troca* (*swap area*), embora armazene páginas. No caso de um disco exclusivo ou uma partição de disco, essa área geralmente é formatada usando uma estrutura de sistema de arquivos otimizada para a transferência rápida das páginas.

Páginas que foram transferidas da memória para o disco possivelmente serão acessadas no futuro por seus processos. Quando um processo tentar acessar uma página que está em disco, esta deve ser transferida de volta para a memória para possibilitar o acesso, de forma transparente ao processo. Conforme exposto na Seção 15.6, quando um processo acessa uma página, a MMU verifica se a mesma está presente na memória RAM; em caso positivo, faz o acesso ao endereço físico correspondente. Caso contrário, a MMU gera uma interrupção de falta de página (*page fault*) que desvia a execução para o sistema operacional.

O sistema operacional verifica se a página é válida, usando os *flags* de controle da tabela de páginas. Caso a página seja inválida, o processo tentou acessar um endereço

inválido e deve ser abortado. Por outro lado, caso a página solicitada seja válida, o processo deve ser suspenso enquanto o sistema transfere a página de volta para a memória RAM e faz os ajustes necessários na tabela de páginas. Uma vez a página carregada em memória, o processo pode continuar sua execução. O fluxograma da Figura 17.3 apresenta as principais ações desenvolvidas pelo mecanismo de paginação em disco.

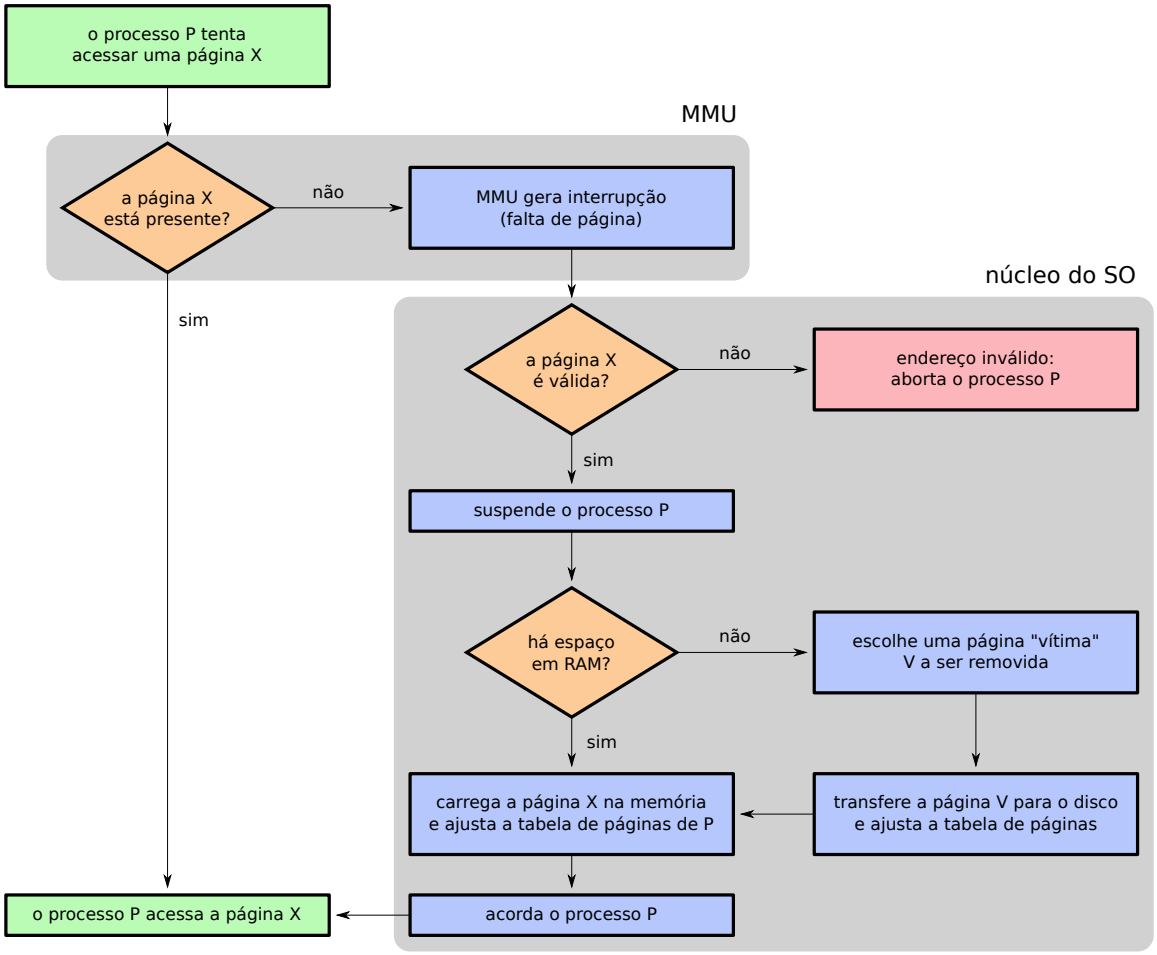

Figura 17.3: Ações do mecanismo de paginação em disco.

Caso a memória principal já esteja cheia, uma página deverá ser movida para o disco antes de trazer de volta a página faltante. Isso implica em mais operações de leitura e escrita no disco e portanto em mais demora para atender o pedido do processo. Muitos sistemas, como o Linux e o Solaris, evitam essa situação mantendo um *daemon*[1] responsável por escolher e transferir páginas para o disco, sempre que a quantidade de memória livre estiver abaixo de um certo limiar.

Retomar a execução do processo que gerou a falta de página pode ser uma tarefa complexa. Como a instrução que gerou a falta de página não foi completada, ela deve ser reexecutada. No caso de instruções simples, envolvendo apenas um endereço de memória sua reexecução é trivial. Todavia, no caso de instruções que envolvam várias ações e vários endereços de memória, deve-se descobrir qual dos endereços gerou a

[1] *Daemons* são processos que executam continuamente, sem interação com o usuário, para prover serviços ao sistema operacional, como gestão de impressoras, de conexões de rede, etc. A palavra *daemon* vem do grego antigo e significa "ser sobrenatural" ou "espírito".

falta de página, que ações da instrução foram executadas e então executar somente o que estiver faltando. A maioria dos processadores atuais provê registradores especiais que auxiliam nessa tarefa.

17.2.2 Eficiência

O mecanismo de paginação em disco permite usar o disco como uma extensão de memória RAM, de forma transparente para os processos. Seria a solução ideal para as limitações da memória principal, se não houvesse um problema importante: o tempo de acesso dos discos utilizados. Conforme os valores indicados na Tabela 14.1, um disco rígido típico tem um tempo de acesso cerca de 100.000 vezes maior que a memória RAM. Cada falta de página provocada por um processo implica em um acesso ao disco, para buscar a página faltante (ou dois acessos, caso a memória RAM esteja cheia e outra página tenha de ser removida antes). Assim, faltas de página muito frequentes irão gerar muitos acessos ao disco, aumentando o tempo médio de acesso à memória e, em consequência, diminuindo o desempenho geral do sistema.

Para demonstrar o impacto das faltas de página no desempenho, consideremos um sistema cuja memória RAM tem um tempo de acesso de 60 ns ($60 \times 10^{-9}s$) e cujo disco de troca tem um tempo de acesso de 6 ms ($6 \times 10^{-3}s$), no qual ocorre uma falta de página a cada milhão de acessos (10^6 acessos). Caso a memória não esteja saturada, o tempo médio de acesso será:

$$\begin{aligned} t_{médio} &= \frac{(999.999 \times 60ns) + 6ms + 60ns}{1.000.000} \\ &= \frac{10^6 \times 60 \times 10^{-9} + 6 \times 10^{-3}}{10^6} \\ t_{médio} &= 66ns \end{aligned}$$

Caso a memória esteja saturada, o tempo médio será maior:

$$\begin{aligned} t_{médio} &= \frac{(999.999 \times 60ns) + 2 \times 6ms + 60ns}{1.000.000} \\ &= \frac{10^6 \times 60 \times 10^{-9} + 2 \times 6 \times 10^{-3}}{10^6} \\ t_{médio} &= 72ns \end{aligned}$$

Caso a frequência de falta de páginas aumente para uma falta a cada 100.000 acessos (10^5 acessos), o tempo médio de acesso à memória subirá para 120 ns no primeiro caso (memória não saturada) e 180 ns no segundo caso (memória saturada).

A frequência de faltas de página depende de vários fatores, como:

- O tamanho da memória RAM, em relação à demanda dos processos em execução: sistemas com memória insuficiente, ou muito carregados, podem gerar muitas faltas de página, prejudicando o seu desempenho e podendo ocasionar o fenômeno conhecido como *thrashing* (Seção 17.7).

- o comportamento dos processos em relação ao uso da memória: processos que agrupem seus acessos a poucas páginas em cada momento, respeitando a localidade de referências (Seção 15.8), necessitam usar menos páginas simultaneamente e geram menos faltas de página.
- A escolha das páginas a remover da memória: caso sejam removidas páginas usadas com muita frequência, estas serão provavelmente acessadas pouco tempo após sua remoção, gerando mais faltas de página. A escolha das páginas a remover é responsabilidade dos algoritmos apresentados na Seção 17.3.

17.2.3 Critérios de seleção

Vários critérios podem ser usados para escolher páginas "vítimas", ou seja, páginas a transferir da memória RAM para o armazenamento secundário:

Idade da página: há quanto tempo a página está na memória; páginas muito antigas talvez sejam pouco usadas.

Frequência de acessos à página: páginas muito acessadas em um passado recente possivelmente ainda o serão em um futuro próximo.

Data do último acesso: páginas há mais tempo sem acessar possivelmente serão pouco acessadas em um futuro próximo (sobretudo se os processos respeitarem o princípio da localidade de referências).

Prioridade do processo proprietário: processos de alta prioridade, ou de tempo real, podem precisar de suas páginas de memória rapidamente; se elas estiverem no disco, seu desempenho ou tempo de resposta poderão ser prejudicados.

Conteúdo da página: páginas cujo conteúdo seja código executável exigem menos esforço do mecanismo de paginação, porque seu conteúdo já está mapeado no disco (dentro do arquivo executável correspondente ao processo). Por outro lado, páginas de dados que tenham sido alteradas precisam ser salvas na área de troca.

Páginas especiais: páginas contendo *buffers* de operações de entrada/saída podem ocasionar dificuldades ao núcleo caso não estejam na memória no momento em que ocorrer a transferência de dados entre o processo e o dispositivo físico. O processo também pode solicitar que certas páginas contendo informações sensíveis (como senhas ou chaves criptográficas) não sejam copiadas na área de troca, por segurança.

A escolha correta das páginas a retirar da memória física é um fator essencial para a eficiência do mecanismo de paginação. Más escolhas poderão remover da memória páginas muito usadas, aumentando a taxa de faltas de página e diminuindo o desempenho do sistema.

17.3 Algoritmos clássicos

Existem vários algoritmos para a escolha de páginas a substituir na memória, visando reduzir a frequência de falta de páginas, que levam em conta alguns dos fatores acima enumerados. Os principais algoritmos serão apresentados na sequência.

17.3.1 Cadeia de referências

Uma ferramenta importante para o estudo dos algoritmos de substituição de páginas é a *cadeia de referências* (*reference string*), que indica a sequência de páginas acessadas por um processo ao longo de sua execução, considerando todos os endereços acessados pelo processo nas várias áreas de memória que o compõem (código, dados, pilha, *heap*, etc). Ao submeter a cadeia de referências de uma execução aos vários algoritmos, podemos calcular quantas faltas de página cada um geraria naquela execução em particular, permitindo assim comparar suas eficiências.

Cadeias de referências de execuções reais podem ser muito longas: considerando um tempo de acesso à memória de 50 ns, em apenas um segundo de execução ocorrem por volta de 20 milhões de acessos à memória. Além disso, a obtenção de cadeias de referências confiáveis é uma área de pesquisa importante, por envolver técnicas complexas de coleta, filtragem e compressão de dados de execução de sistemas [Uhlig and Mudge, 1997]. Para possibilitar a comparação dos algoritmos de substituição de páginas apresentados na sequência, será usada a seguinte cadeia de referências fictícia, obtida de [Silberschatz et al., 2001]:

$$7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$$

Deve-se observar que acessos consecutivos a uma mesma página não são relevantes para a análise dos algoritmos, porque somente o primeiro acesso em cada grupo de acessos consecutivos provoca uma falta de página.

17.3.2 Algoritmo Ótimo

Idealmente, a melhor página a remover da memória em um dado instante é aquela que ficará mais tempo sem ser usada pelos processos. Esta ideia simples define o *algoritmo ótimo* (OPT). Entretanto, como o comportamento futuro dos processos não pode ser previsto com precisão, este algoritmo não é implementável. Mesmo assim ele é importante, porque define um limite mínimo conceitual: se, para uma dada cadeia de referências, o algoritmo ótimo gera X faltas de página, nenhum outro algoritmo irá gerar menos de X faltas de página ao tratar essa mesma cadeia. Assim, seu resultado serve como parâmetro para a avaliação dos demais algoritmos.

A aplicação do algoritmo ótimo à cadeia de referências apresentada na Seção anterior, considerando uma memória física com 3 quadros, é apresentada na Tabela 17.1. Nesse caso, o algoritmo OPT gera 9 faltas de página.

17.3.3 Algoritmo FIFO

Um critério simples e factível a considerar para a escolha das páginas a substituir poderia ser sua "idade", ou seja, o tempo em que estão na memória. Assim, páginas mais antigas podem ser removidas para dar lugar a novas páginas. Esse algoritmo é muito simples de implementar: basta organizar as páginas em uma fila de números de páginas com política FIFO (*First In, First Out*). Os números das páginas recém carregadas na memória são registrados no final da lista, enquanto os números das próximas páginas a substituir na memória são obtidos no início da lista.

A aplicação do algoritmo FIFO à cadeia de referências apresentada na Seção anterior, considerando uma memória física com 3 quadros, é apresentada na Tabela 17.2. Nesse caso, o algoritmo gera no total 15 faltas de página, 6 a mais que o algoritmo ótimo.

t	página acessada	quadros q_o	q_1	q_2	falta de página?	ação realizada
0						situação inicial, quadros vazios
1	7	7			✓	p_7 é carregada em q_0
2	0	7	0		✓	p_0 é carregada em q_1
3	1	7	0	1	✓	p_1 é carregada em q_2
4	2	2	0	1	✓	p_2 substitui p_7 (que só será acessada em $t = 18$)
5	0	2	0	1		p_0 já está na memória
6	3	2	0	3	✓	p_3 substitui p_1
7	0	2	0	3		p_0 já está na memória
8	4	2	4	3	✓	p_4 substitui p_0
9	2	2	4	3		p_2 já está na memória
10	3	2	4	3		p_3 já está na memória
11	0	2	0	3	✓	p_0 substitui p_4
12	3	2	0	3		p_3 já está na memória
13	2	2	0	3		p_2 já está na memória
14	1	2	0	1	✓	p_1 substitui p_3
15	2	2	0	1		p_2 já está na memória
16	0	2	0	1		p_0 já está na memória
17	1	2	0	1		p_1 já está na memória
18	7	7	0	1	✓	p_7 substitui p_2
19	0	7	0	1		p_0 já está na memória
20	1	7	0	1		p_1 já está na memória

Tabela 17.1: Aplicação do algoritmo de substituição ótimo.

Apesar de ter uma implementação simples, na prática este algoritmo não oferece bons resultados. Seu principal defeito é considerar somente a idade da página, sem levar em conta sua importância. Páginas carregadas na memória há muito tempo podem estar sendo frequentemente acessadas, como é o caso de áreas de memória contendo bibliotecas dinâmicas compartilhadas por muitos processos, ou páginas de processos servidores lançados durante a inicialização (*boot*) da máquina.

17.3.4 Algoritmo LRU

Uma aproximação implementável do algoritmo ótimo é proporcionada pelo algoritmo LRU (*Least Recently Used*, menos recentemente usado). Neste algoritmo, a escolha recai sobre as páginas que estão na memória há mais tempo **sem ser acessadas**. Assim, páginas antigas e menos usadas são as escolhas preferenciais. Páginas antigas mas de uso frequente não são penalizadas por este algoritmo, ao contrário do que ocorre no algoritmo FIFO. Pode-se observar facilmente que este algoritmo é simétrico do algoritmo OPT em relação ao tempo: enquanto o OPT busca as páginas que serão acessadas "mais longe" no futuro do processo, o algoritmo LRU busca as páginas que foram acessadas "mais longe" no seu passado.

t	página acessada	quadros q_o	q_1	q_2	falta de página?	ação realizada
0						situação inicial, quadros vazios
1	7	7			✓	p_7 é carregada em q_0
2	0	7	0		✓	p_0 é carregada em q_1
3	1	7	0	1	✓	p_1 é carregada em q_2
4	2	2	0	1	✓	p_2 substitui p_7 (carregada em $t = 1$)
5	0	2	0	1		p_0 já está na memória
6	3	2	3	1	✓	p_3 substitui p_0
7	0	2	3	0	✓	p_1 substitui p_1
8	4	4	3	0	✓	p_4 substitui p_2
9	2	4	2	0	✓	p_2 substitui p_3
10	3	4	2	3	✓	p_3 substitui p_0
11	0	0	2	3	✓	p_0 substitui p_4
12	3	0	2	3		p_3 já está na memória
13	2	0	2	3		p_2 já está na memória
14	1	0	1	3	✓	p_1 substitui p_2
15	2	0	1	2	✓	p_2 substitui p_3
16	0	0	1	2		p_0 já está na memória
17	1	0	1	2		p_1 já está na memória
18	7	7	1	2	✓	p_7 substitui p_0
19	0	7	0	2	✓	p_0 substitui p_1
20	1	7	0	1	✓	p_1 substitui p_2

Tabela 17.2: Aplicação do algoritmo de substituição FIFO.

A aplicação do algoritmo LRU à cadeia de referências apresentada na Seção anterior, considerando uma memória física com 3 quadros, é apresentada na Tabela 17.3. Nesse caso, o algoritmo gera 12 faltas de página (três faltas a mais que o algoritmo ótimo).

O algoritmo LRU parte do pressuposto que páginas recentemente acessadas no passado provavelmente serão acessadas em um futuro próximo, e então evita removê-las da memória. Esta hipótese geralmente se verifica na prática, sobretudo se os processos respeitam o princípio da localidade de referência (Seção 15.8). Todavia, o desempenho do algoritmo LRU é prejudicado no caso de acessos com um padrão fortemente sequencial, ou seja, um certo número de páginas são acessadas em sequência e repetidamente (por exemplo: $p_1, p_2, p_3, \cdots, p_n, p_1, p_2, p_3, \cdots p_n, \cdots$). Nessa situação, o desempenho do algoritmo LRU será similar ao do FIFO.

17.3.5 Algoritmo RANDOM

Um algoritmo interessante consiste em escolher aleatoriamente as páginas a retirar da memória. O algoritmo aleatório pode ser útil em situações onde as abordagens LRU e FIFO tem desempenho ruim, como os padrões de acesso fortemente sequenciais

t	página acessada	quadros q_o	q_1	q_2	falta de página?	ação realizada
0						situação inicial, quadros vazios
1	7	7			✓	p_7 é carregada em q_0
2	0	7	0		✓	p_0 é carregada em q_1
3	1	7	0	1	✓	p_1 é carregada em q_2
4	2	2	0	1	✓	p_2 substitui p_7 (há mais tempo sem acesso)
5	0	2	0	1		p_0 já está na memória
6	3	2	0	3	✓	p_3 substitui p_1
7	0	2	0	3		p_0 já está na memória
8	4	4	0	3	✓	p_4 substitui p_2
9	2	4	0	2	✓	p_2 substitui p_3
10	3	4	3	2	✓	p_3 substitui p_0
11	0	0	3	2	✓	p_0 substitui p_4
12	3	0	3	2		p_3 já está na memória
13	2	0	3	2		p_2 já está na memória
14	1	1	3	2	✓	p_1 substitui p_0
15	2	1	3	2		p_2 já está na memória
16	0	1	0	2	✓	p_0 substitui p_3
17	1	1	0	2		p_1 já está na memória
18	7	1	0	7	✓	p_7 substitui p_2
19	0	1	0	7		p_0 já está na memória
20	1	1	0	7		p_1 já está na memória

Tabela 17.3: Aplicação do algoritmo de substituição LRU.

discutidos na seção anterior. De fato, alguns sistemas operacionais mais antigos usavam essa abordagem em situações onde os demais algoritmos funcionavam mal.

17.3.6 Comparação entre algoritmos

O gráfico da Figura 17.4 permite a comparação dos algoritmos OPT, FIFO, LRU e RANDOM sobre a cadeia de referências apresentada na Seção 17.3.1, em função do número de quadros existentes na memória física. Pode-se observar que o melhor desempenho é do algoritmo OPT, enquanto o pior desempenho é proporcionado pelo algoritmo RANDOM.

A Figura 17.5 permite comparar o desempenho desses mesmos algoritmos de substituição de páginas em um cenário mais realista. A cadeia de referências usada corresponde a uma execução do compilador GCC (`http://gcc.gnu.org`) com 10^6 referências a 1.260 páginas distintas. Pode-se perceber as mesmas relações entre os desempenhos dos algoritmos.

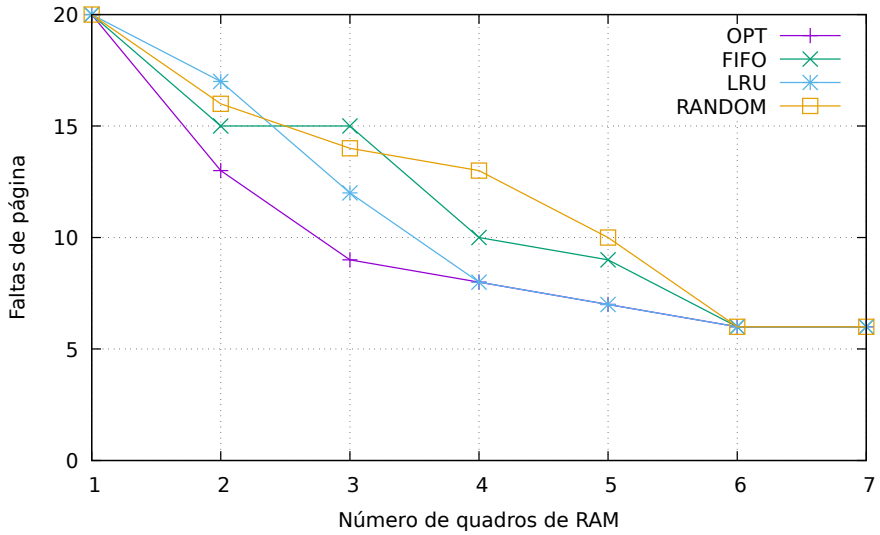

Figura 17.4: Comparação dos algoritmos de substituição de páginas.

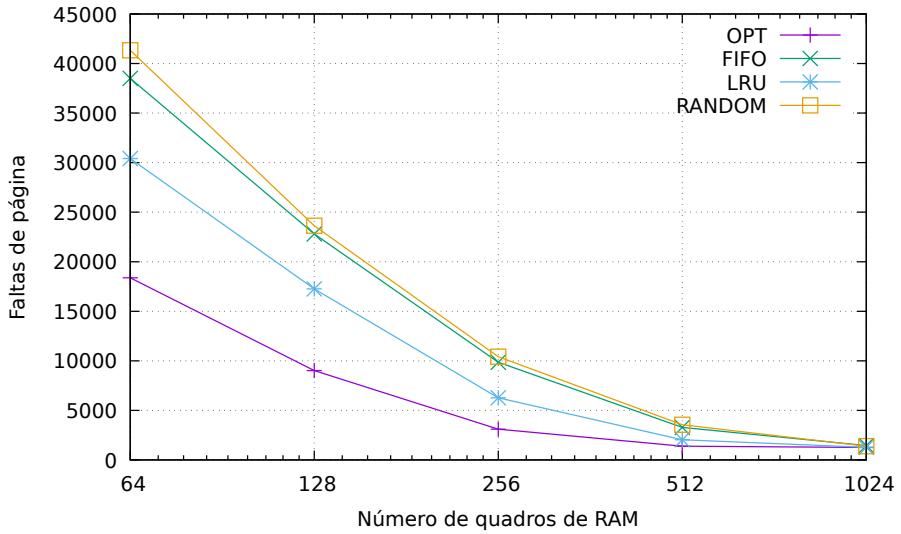

Figura 17.5: Comparação dos algoritmos, usando uma execução do compilador GCC.

17.4 Aproximações do algoritmo LRU

Embora possa ser implementado, o algoritmo LRU completo é pouco usado na prática, porque sua implementação exigiria registrar as datas de acesso às páginas a cada leitura ou escrita na memória, o que é difícil de implementar de forma eficiente em software e com custo proibitivo para implementar em hardware. Além disso, sua implementação exigiria varrer as datas de acesso de todas as páginas para buscar a página com acesso mais antigo (ou manter uma lista de páginas ordenadas por data de acesso), o que exigiria muito processamento. Portanto, a maioria dos sistemas operacionais reais implementa algoritmos baseados em aproximações do LRU.

Esta seção apresenta alguns algoritmos simples que permitem se aproximar do comportamento LRU. Por sua simplicidade, esses algoritmos têm desempenho limitado e por isso somente são usados em sistemas operacionais mais simples. Como exemplos

de algoritmos de substituição de páginas mais sofisticados e com maior desempenho podem ser citados o LIRS [Jiang and Zhang, 2002] e o ARC [Bansal and Modha, 2004].

17.4.1 Algoritmo da segunda chance

O algoritmo FIFO (Seção 17.3.3) move para a área de troca as páginas há mais tempo na memória, sem levar em conta seu histórico de acessos. Uma melhoria simples desse algoritmo consiste em analisar o bit de referência (Seção 15.6.2) de cada página candidata, para saber se ela foi acessada recentemente. Caso tenha sido, essa página recebe uma "segunda chance", voltando para o fim da fila com seu bit de referência ajustado para zero. Dessa forma, evita-se substituir páginas antigas mas muito acessadas. Todavia, caso todas as páginas sejam muito acessadas, o algoritmo vai varrer todas as páginas, ajustar todos os bits de referência para zero e acabará por escolher a primeira página da fila, como faria o algoritmo FIFO.

Uma forma eficiente de implementar este algoritmo é através de uma lista circular de números de página, ordenados de acordo com seu ingresso na memória. Um ponteiro percorre a lista sequencialmente, analisando os bits de referência das páginas e ajustando-os para zero à medida em que avança. Quando uma página vítima é encontrada, ela é movida para o disco e a página desejada é carregada na memória no lugar da vítima, com seu bit de referência ajustado para um (pois acaba de ser acessada). Essa implementação é conhecida como *algoritmo do relógio* e pode ser vista na Figura 17.6.

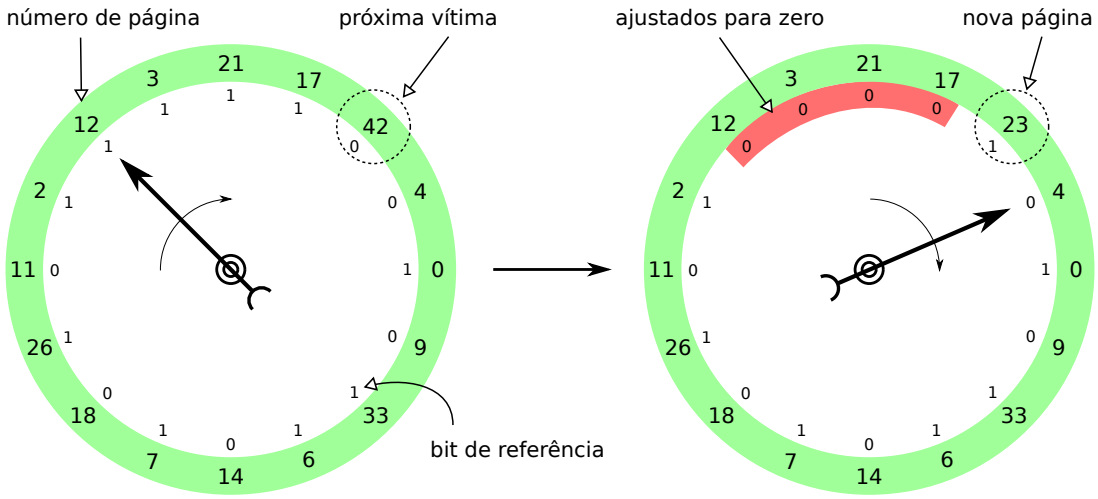

Figura 17.6: Algoritmo da segunda chance (ou do relógio).

17.4.2 Algoritmo NRU

O algoritmo da segunda chance leva em conta somente o bit de referência de cada página ao escolher as vítimas para substituição. O algoritmo NRU (*Not Recently Used*, ou *não usada recentemente*) melhora essa escolha, ao considerar também o bit de modificação (*dirty bit*, vide Seção 15.6.2), que indica se o conteúdo de uma página foi modificado após ela ter sido carregada na memória.

Usando os bits R (referência) e M (modificação), é possível classificar as páginas em memória em quatro níveis de importância:

- 00 ($R = 0, M = 0$): páginas que não foram referenciadas recentemente e cujo conteúdo não foi modificado. São as melhores candidatas à substituição, pois podem ser simplesmente retiradas da memória.
- 01 ($R = 0, M = 1$): páginas que não foram referenciadas recentemente, mas cujo conteúdo já foi modificado. Não são escolhas tão boas, porque terão de ser gravadas na área de troca antes de serem substituídas.
- 10 ($R = 1, M = 0$): páginas referenciadas recentemente, cujo conteúdo permanece inalterado. São provavelmente páginas de código que estão sendo usadas ativamente e serão referenciadas novamente em breve.
- 11 ($R = 1, M = 1$): páginas referenciadas recentemente e cujo conteúdo foi modificado. São a pior escolha, porque terão de ser gravadas na área de troca e provavelmente serão necessárias em breve.

O algoritmo NRU consiste simplesmente em tentar substituir primeiro páginas do nível 0; caso não encontre, procura candidatas no nível 1 e assim sucessivamente. Pode ser necessário percorrer várias vezes a lista circular até encontrar uma página adequada para substituição.

17.4.3 Algoritmo do envelhecimento

Outra possibilidade de melhoria do algoritmo da segunda chance consiste em usar os bits de referência das páginas para construir *contadores de acesso* às mesmas. A cada página é associado um contador inteiro com N bits (geralmente 8 bits são suficientes). Periodicamente, o algoritmo varre as tabelas de páginas, lê os bits de referência e agrega seus valores aos contadores de acessos das respectivas páginas. Uma vez lidos, os bits de referência são ajustados para zero, para registrar as referências de páginas que ocorrerão durante próximo período.

O valor lido de cada bit de referência não deve ser simplesmente somado ao contador, por duas razões: o contador chegaria rapidamente ao seu valor máximo (*overflow*) e a simples soma não permitiria diferenciar acessos recentes dos mais antigos. Por isso, outra solução foi encontrada: cada contador é deslocado para a direita 1 bit, descartando o bit menos significativo (LSB - *Least Significant Bit*). Em seguida, o valor do bit de referência é colocado na primeira posição à esquerda do contador, ou seja, em seu bit mais significativo (MSB - *Most Significant Bit*). Dessa forma, acessos mais recentes têm um peso maior que acessos mais antigos, e o contador nunca ultrapassa seu valor máximo.

O exemplo a seguir mostra a evolução dos contadores para quatro páginas distintas, usando os valores dos respectivos bits de referência R. Os valores decimais dos contadores estão indicados entre parênteses, para facilitar a comparação. Observe que as páginas acessadas no último período (p_2 e p_4) têm seus contadores aumentados, enquanto aquelas não acessadas (p_1 e p_3) têm seus contadores diminuídos.

	R		*contadores*			R		*contadores*	
p_1	0		0000 0011	(3)		0		**0**000 0001	(1)
p_2	1	com	0011 1101	(61)	$\Longrightarrow$	0	e	**1**001 1110	(158)
p_3	0		1010 1000	(168)		0		**0**101 0100	(84)
p_4	1		1110 0011	(227)		0		**1**111 0001	(241)

O contador construído por este algoritmo constitui uma aproximação razoável do algoritmo LRU: páginas menos acessadas "envelhecerão", ficando com contadores menores, enquanto páginas mais acessadas permanecerão "jovens", com contadores maiores. Por essa razão, esta estratégia é conhecida como *algoritmo do envelhecimento* [Tanenbaum, 2003], ou *algoritmo dos bits de referência adicionais* [Silberschatz et al., 2001].

17.5 Conjunto de trabalho

A localidade de referências (estudada na Seção 15.8) mostra que os processos normalmente acessam apenas uma pequena fração de suas páginas a cada instante. O conjunto de páginas acessadas na história recente de um processo é chamado *Conjunto de Trabalho* (*Working Set*, ou *ws*) [Denning, 1980, 2006]. A composição do conjunto de trabalho é dinâmica, variando à medida em que o processo executa e evolui seu comportamento, acessando novas páginas e deixando de acessar outras. Para ilustrar esse conceito, consideremos a cadeia de referências apresentada na Seção 17.3.1. Considerando como história recente as últimas n páginas acessadas pelo processo, a evolução do conjunto de trabalho *ws* do processo que gerou aquela cadeia é apresentada na Tabela 17.4.

t	**página**	$ws(n = 3)$	$ws(n = 4)$	$ws(n = 5)$
1	7	7	7	7
2	0	0, 7	0, 7	0, 7
3	1	0, 1, 7	0, 1, 7	0, 1, 7
4	2	0, 1, 2	0, 1, 2, 7	0, 1, 2, 7
5	0	0, 1, 2	0, 1, 2	0, 1, 2, 7
6	3	0, 2, 3	0, 1, 2, 3	0, 1, 2, 3
7	0	0, 3	0, 2, 3	0, 1, 2, 3
8	4	0, 3, 4	0, 3, 4	0, 2, 3, 4
9	2	0, 2, 4	0, 2, 3, 4	0, 2, 3, 4
10	3	2, 3, 4	0, 2, 3, 4	0, 2, 3, 4
11	0	0, 2, 3	0, 2, 3, 4	0, 2, 3, 4
12	3	0, 3	0, 2, 3	0, 2, 3, 4
13	2	0, 2, 3	0, 2, 3	0, 2, 3
14	1	1, 2, 3	0, 1, 2, 3	0, 1, 2, 3
15	2	1, 2	1, 2, 3	0, 1, 2, 3
16	0	0, 1, 2	0, 1, 2	0, 1, 2, 3
17	1	0, 1, 2	0, 1, 2	0, 1, 2
18	7	0, 1, 7	0, 1, 2, 7	0, 1, 2, 7
19	0	0, 1, 7	0, 1, 7	0, 1, 2, 7
20	1	0, 1, 7	0, 1, 7	0, 1, 7

Tabela 17.4: Conjuntos de trabalho *ws* para $n = 3$, $n = 4$ e $n = 5$.

O tamanho e a composição do conjunto de trabalho dependem do número de páginas consideradas em sua história recente (o valor n na Tabela 17.4). Em sistemas reais, essa dependência não é linear, mas segue uma proporção exponencial inversa, devido à localidade de referências. Por essa razão, a escolha precisa do tamanho da história recente a considerar não é crítica. Esse fenômeno pode ser observado na Tabela 17.4: assim que a localidade de referências se torna mais forte (a partir de $t = 12$), os três conjuntos de trabalho ficam muito similares. Outro exemplo é apresentado na Figura 17.7, que mostra o tamanho médio dos conjuntos de trabalhos observados na execução do programa *gThumb* (analisado na Seção 15.8), em função do tamanho da história recente considerada (em número de páginas referenciadas).

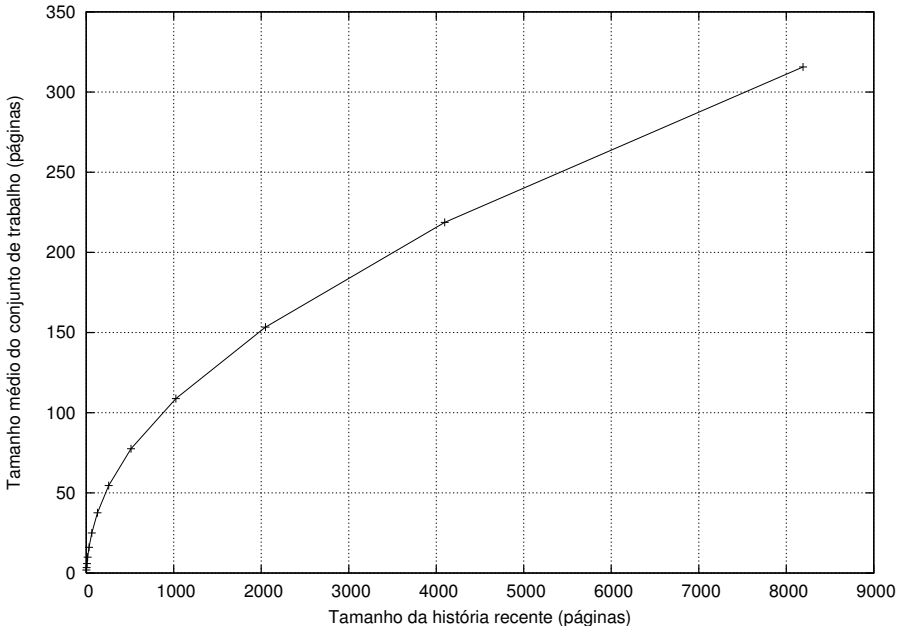

Figura 17.7: Tamanho do conjunto de trabalho do programa *gThumb*.

Se um processo tiver todas as páginas de seu conjunto de trabalho carregadas na memória, ele sofrerá poucas faltas de página, pois somente acessos a novas páginas poderão gerar faltas. Essa constatação permite delinear um algoritmo simples para substituição de páginas: só substituir páginas que não pertençam ao conjunto de trabalho de nenhum processo ativo. Contudo, esse algoritmo é difícil de implementar, pois exigiria manter atualizado o conjunto de trabalho de cada processo a cada acesso à memória, o que teria um custo computacional proibitivo.

Uma alternativa mais simples e eficiente de implementar seria verificar que páginas cada processo acessou recentemente, usando a informação dos respectivos bits de referência. Essa é a base do algoritmo *WSClock* (*Working Set Clock*) [Carr and Hennessy, 1981], uma modificação do algoritmo do relógio (Seção 17.4.1). Como no algoritmo do relógio, as páginas carregadas na memória também são organizadas em uma lista circular, por ordem de instante de carga na memória. Cada página p dessa lista tem uma data de último acesso $t_a(p)$.

No *WSClock*, define-se um prazo de validade τ para as páginas, entre dezenas e centenas de milissegundos; a idade $i(p)$ de uma página p na memória é definida como a diferença entre a data de seu último acesso $t_a(p)$ e o instante atual t_{now} ($i(p) = t_{now} - t_a(p)$).

Quando há necessidade de substituir páginas, o ponteiro percorre a lista buscando páginas "vítimas":

1. Ao encontrar uma página p referenciada (com $R(p) = 1$), a data de seu último acesso é atualizada ($t_a(p) = t_{now}$), seu bit de referência é limpo ($R(p) = 0$) e o ponteiro do relógio avança, ignorando-a.
2. Ao encontrar uma página p não-referenciada (com $R(p) = 0$):
   (a) se a idade de p for válida ($i(p) \leq \tau$), a página está no conjunto de trabalho e deve ser ignorada;
   (b) caso contrário ($i(p) > \tau$), p está fora do conjunto de trabalho. Neste caso:
      i. Se $M(p) = 0$, a página p não foi modificada e pode ser substituída;
      ii. caso contrário ($M(p) = 1$), agenda-se uma escrita dessa página em disco e o ponteiro do relógio avança, ignorando-a.
3. Caso o ponteiro dê uma volta completa na lista e não encontre página com $i(p) > \tau$, $R = 0$ e $M = 0$:
   (a) substituir a página mais antiga (com o menor $t_a(p)$) com $R = 0$ e $M = 0$;
   (b) se não achar, substituir a página mais antiga com $M = 0$ e $R = 1$;
   (c) se não achar, substituir a página mais antiga com $R = 0$;
   (d) se não achar, substituir a página mais antiga.

O algoritmo *WSClock* pode ser implementado de forma eficiente, porque a data último acesso de cada página não precisa ser atualizada a cada acesso à memória, mas somente quando a referência da página na lista circular é visitada pelo ponteiro do relógio (caso $R = 1$). Todavia, esse algoritmo não é uma implementação "pura" do conceito de conjunto de trabalho, mas uma composição de conceitos de vários algoritmos: FIFO e segunda chance (estrutura e percurso do relógio), Conjuntos de trabalho (divisão das páginas em dois grupos conforme sua idade), LRU (escolha das páginas com datas de acesso mais antigas) e NRU (preferência às páginas não modificadas).

17.6 A anomalia de Belady

Espera-se que, quanto mais memória física um sistema possua, menos faltas de página ocorram. Todavia, esse comportamento intuitivo não se verifica em todos os algoritmos de substituição de páginas. Alguns algoritmos, como o FIFO, podem apresentar um comportamento estranho: ao aumentar o número de quadros de memória, o número de faltas de página geradas pelo algoritmo aumenta, ao invés de diminuir. Esse comportamento atípico de alguns algoritmos foi estudado pelo matemático húngaro Laslo Belady nos anos 60, sendo por isso denominado *anomalia de Belady*.

A seguinte cadeia de referências permite observar esse fenômeno; o comportamento dos algoritmos OPT, FIFO e LRU ao processar essa cadeia pode ser visto na Figura 17.8, que exibe o número de faltas de página em função do número de quadros de memória disponíveis no sistema. A anomalia pode ser observada no algoritmo FIFO:

ao aumentar a memória de 4 para 5 quadros, esse algoritmo passa de 22 para 24 faltas de página.

$$0, 1, 2, 3, 4, 0, 1, 2, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 0, 1, 2, 5, 0, 1, 2, 3, 4, 5$$

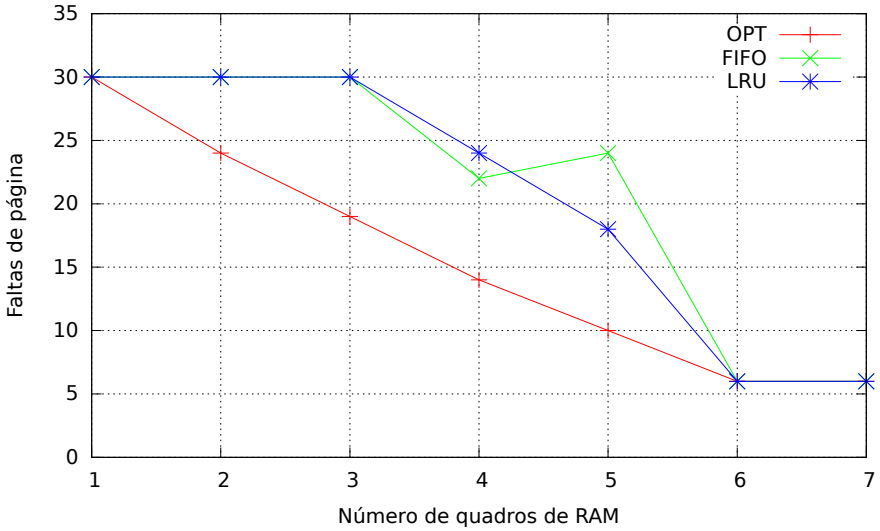

Figura 17.8: A anomalia de Belady.

Estudos demonstraram que uma família de algoritmos denominada *algoritmos de pilha* (à qual pertencem os algoritmos OPT e LRU, entre outros) não apresenta a anomalia de Belady [Tanenbaum, 2003].

17.7 Thrashing

Na Seção 17.2.2, foi demonstrado que o tempo médio de acesso à memória RAM aumenta significativamente à medida em que aumenta a frequência de faltas de página. Caso a frequência de faltas de páginas seja muito elevada, o desempenho do sistema como um todo pode ser severamente prejudicado.

Conforme discutido na Seção 17.5, cada processo tem um conjunto de trabalho, ou seja, um conjunto de páginas que devem estar na memória para sua execução naquele momento. Se o processo tiver uma boa localidade de referência, esse conjunto é pequeno e varia lentamente. Caso a localidade de referência seja ruim, o conjunto de trabalho geralmente é grande e muda rapidamente. Enquanto houver espaço na memória RAM para os conjuntos de trabalho dos processos ativos, não haverão problemas. Contudo, caso a soma de todos os conjuntos de trabalho dos processos prontos para execução seja maior que a memória RAM disponível no sistema, poderá ocorrer um fenômeno conhecido como *thrashing* [Denning, 1980, 2006].

No *thrashing*, a memória RAM não é suficiente para todos os processos ativos, portanto muitos processos não conseguem ter seus conjuntos de trabalho totalmente carregados na memória. Cada vez que um processo recebe o processador, executa algumas instruções, gera uma falta de página e volta ao estado suspenso, até que a página faltante seja trazida de volta à RAM. Todavia, para trazer essa página à RAM será necessário abrir espaço na memória, transferindo algumas páginas (de outros processos)

para o disco. Quanto mais processos estiverem nessa situação, maior será a atividade de paginação e maior o número de processos no estado suspenso, aguardando páginas.

A Figura 17.9 ilustra o conceito de *thrashing*: ela mostra a taxa de uso do processador (quantidade de processos na fila de prontos) em função do número de processos ativos no sistema. Na zona à esquerda não há *thrashing*, portanto a taxa de uso do processador aumenta com o aumento do número de processos. Caso esse número aumente muito, a memória RAM não será suficiente para todos os conjuntos de trabalho e o sistema entra em uma situação de *thrashing*: muitos processos passarão a ficar suspensos aguardando a paginação, diminuindo a taxa de uso do processador. Quanto mais processos ativos, menos o processador será usado e mais lento ficará o sistema. Pode-se observar que um sistema ideal com memória infinita não teria esse problema, pois sempre haveria memória suficiente para todos os processos ativos.

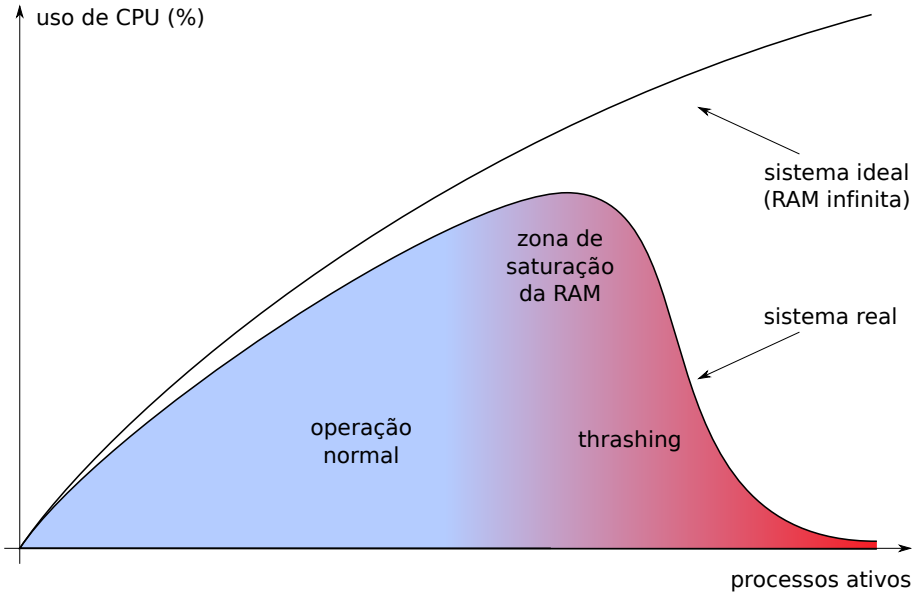

Figura 17.9: Comportamento de um sistema com *thrashing*.

Um sistema operacional sob *thrashing* tem seu desempenho muito prejudicado, a ponto de parar de responder ao usuário e se tornar inutilizável. Por isso, esse fenômeno deve ser evitado. Para tal, pode-se aumentar a quantidade de memória RAM do sistema, limitar a quantidade máxima de processos ativos, ou mudar a política de escalonamento dos processos durante o *thrashing*, para evitar a competição pela memória disponível. Vários sistemas operacionais adotam medidas especiais para situações de *thrashing*, como suspender em massa os processos ativos, adotar uma política de escalonamento de processador que considere o uso da memória, aumentar o *quantum* de processador para cada processo ativo, ou simplesmente abortar os processos com maior alocação de memória ou com maior atividade de paginação.

Exercícios

1. O que é uma falta de página? Quais são suas causa possíveis e como o sistema operacional deve tratá-las?

2. Calcule o tempo médio efetivo de acesso à memória se o tempo de acesso à RAM é de 5 ns, o de acesso ao disco é de 5 ms e em média ocorre uma falta de página a cada 1.000.000 (10^6) de acessos à memória. Considere que a memória RAM sempre tem espaço livre para carregar novas páginas. Apresente e explique seu raciocínio.

3. Repita o exercício anterior, considerando que a memória RAM está saturada: para carregar uma nova página na memória é necessário antes abrir espaço, retirando outra página.

4. Considere um sistema de memória com quatro quadros de RAM e oito páginas a alocar. Os quadros contêm inicialmente as páginas 7, 4 e 1, carregadas em memória nessa sequência. Determine quantas faltas de página ocorrem na sequência de acesso $\{0, 1, 7, 2, 3, 2, 7, 1, 0, 3\}$, para os algoritmos de escalonamento de memória FIFO, OPT e LRU.

5. Repita o exercício anterior considerando um sistema de memória com três quadros de RAM.

6. Um computador tem 8 quadros de memória física; os parâmetros usados pelo mecanismo de paginação em disco são indicados na tabela a seguir:

página	carga na memória	último acesso	bit R	bit M
p_0	14	58	1	1
p_1	97	97	1	0
p_2	124	142	1	1
p_3	47	90	0	1
p_4	29	36	1	0
p_5	103	110	0	0
p_6	131	136	1	1
p_7	72	89	0	0

   Qual será a próxima página a ser substituída, considerando os algoritmos LRU, FIFO, segunda chance e NRU? Indique seu raciocínio.

7. Considere um sistema com 4 quadros de memória. Os seguintes valores são obtidos em dez leituras consecutivas dos bits de referência desses quadros: 0101, 0011, 1110, 1100, 1001, 1011, 1010, 0111, 0110 e 0111. Considerando o algoritmo de envelhecimento, determine o valor final do contador associado a cada página e indique que quadro será substituído.

8. Em um sistema que usa o algoritmo *WSClock*, o conteúdo da fila circular de referências de página em $t_c = 220$ é indicado pela tabela a seguir. Considerando que o ponteiro está em p_0 e que $\tau = 50$, qual será a próxima página a substituir? E no caso de $\tau = 100$?

página	último acesso	bit R	bit M
p_0	142	1	0
p_1	197	0	0
p_2	184	0	1
p_3	46	0	1
p_4	110	0	0
p_5	167	0	1
p_6	97	0	1
p_7	129	1	0

9. Sobre as afirmações a seguir, relativas à gerência de memória, indique quais são incorretas, justificando sua resposta:

   (a) Por "Localidade de referências" entende-se o percentual de páginas de um processo que se encontram na memória RAM.

   (b) De acordo com a anomalia de Belady, o aumento de memória de um sistema pode implicar em pior desempenho.

   (c) A localidade de referência influencia significativamente a velocidade de execução de um processo.

   (d) O algoritmo LRU é implementado na maioria dos sistemas operacionais, devido à sua eficiência e baixo custo computacional.

   (e) O compartilhamento de páginas é implementado copiando-se as páginas a compartilhar no espaço de endereçamento de cada processo.

   (f) O algoritmo ótimo define o melhor comportamento possível em teoria, mas não é implementável.

Atividades

1. Construa um simulador de algoritmos de substituição de páginas. O simulador deve receber como entrada o tamanho da RAM (em quadros) e a sequência de referências a páginas de memória e gerar como saída o número de faltas de página geradas, para os algoritmos OPT, FIFO e LRU.

Referências

S. Bansal and D. Modha. CAR: Clock with adaptive replacement. In *USENIX Conference on File and Storage Technologies*, April 2004.

R. Carr and J. Hennessy. WSclock - a simple and effective algorithm for virtual memory management. In *ACM symposium on Operating systems principles*, 1981.

P. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, 6 (1):64–84, January 1980.

P. J. Denning. The locality principle. In J. Barria, editor, *Communication Networks and Computer Systems*, chapter 4, pages 43–67. Imperial College Press, 2006.

S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Intl Conference on Measurement and Modeling of Computer Systems*, pages 31–42, 2002.

A. Silberschatz, P. Galvin, and G. Gagne. *Sistemas Operacionais – Conceitos e Aplicações*. Campus, 2001.

A. Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.

R. Uhlig and T. Mudge. Trace-driven memory simulation: a survey. *ACM Computing Surveys*, 29(2):128–170, June 1997.