Capítulo 18

Tópicos em gestão de memória

Este capítulo traz tópicos de estudo específicos, que aprofundam ou complementam os temas apresentados nesta parte do livro, mas cuja leitura não é essencial para a compreensão do conteúdo principal da disciplina. Algumas seções deste capítulo podem estar incompletas ou não ter sido revisadas.

18.1 Compartilhamento de memória

A memória RAM é um recurso escasso, que deve ser usado de forma eficiente. Nos sistemas atuais, é comum ter várias instâncias do mesmo programa em execução, como várias instâncias de editores de texto, de navegadores, etc. Em servidores, essa situação pode ser ainda mais frequente, com centenas ou milhares de instâncias do mesmo programa carregadas na memória. Por exemplo, em um servidor de e-mail UNIX, cada cliente que se conecta através de protocolos de rede como POP3 ou IMAP terá um processo correspondente no servidor, para atender suas consultas de e-mail (Figura 18.1). Todos esses processos operam com dados distintos (pois atendem a usuários distintos), mas executam o mesmo código de servidor. Assim, centenas ou milhares de cópias do mesmo código executável poderão coexistir na memória do sistema.

Conforme visto na Seção 15.2, a estrutura típica da memória de um processo contém seções separadas para código, dados, pilha e *heap*. Normalmente, a seção de código não tem seu conteúdo modificado durante a execução, portanto geralmente essa seção é protegida contra escritas (*read-only*). Assim, seria possível *compartilhar* essa seção entre todos os processos que executam o mesmo código, economizando memória RAM.

O compartilhamento de código entre processos pode ser implementado de forma muito simples e transparente para os processos envolvidos, através dos mecanismos de tradução de endereços oferecidos pela MMU, como segmentação e paginação. No caso da segmentação, basta fazer com que os segmentos de código dos processos apontem para o mesmo trecho da memória física, como indica a Figura 18.2. É importante observar que o compartilhamento é transparente para os processos: cada processo continua a acessar endereços lógicos em seu próprio segmento de código.

No caso da paginação, a unidade básica de compartilhamento é a página. Assim, a tabela de páginas de cada processo envolvido é ajustada para referenciar os mesmos quadros de memória física. É importante observar que, embora referenciem os mesmos endereços físicos, as páginas compartilhadas podem ter endereços lógicos distintos. A Figura 18.3 ilustra o compartilhamento de páginas entre dois processos.

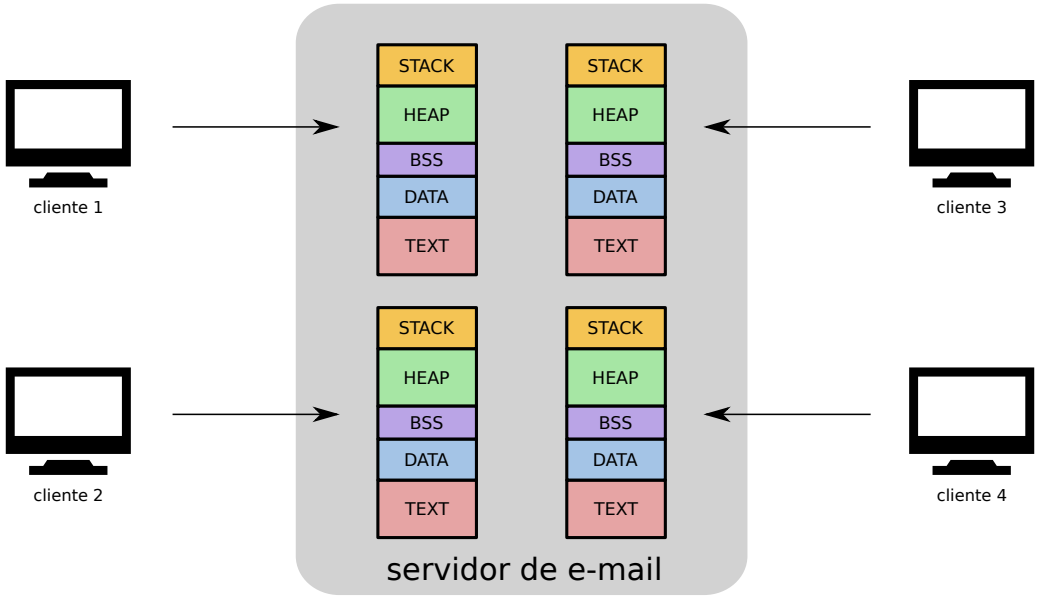

Figura 18.1: Várias instâncias do mesmo processo.

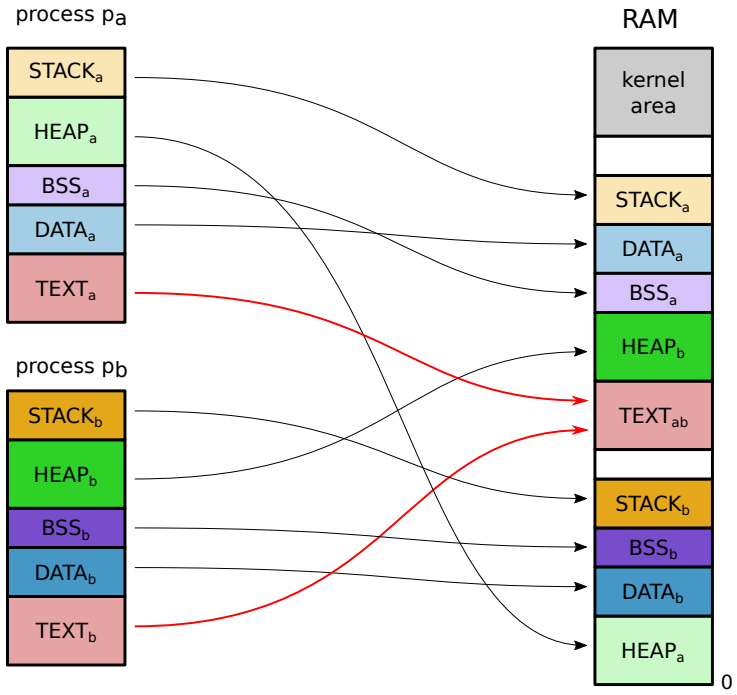

Figura 18.2: Compartilhamento de segmentos.

O compartilhamento das seções de código permite proporcionar uma grande economia no uso da memória física, sobretudo em servidores e sistemas multiusuários. Por exemplo: consideremos um editor de textos que necessite de 100 MB de memória para executar, dos quais 60 MB são ocupados por código executável e bibliotecas. Sem o compartilhamento das seções de código, 10 instâncias do editor consumiriam 1.000 MB de memória; com o compartilhamento, esse consumo cairia para 460 MB: uma área compartilhada com 60 MB contendo o código e mais 10 áreas de 40 MB cada uma, contendo os dados e pilha de cada processo.

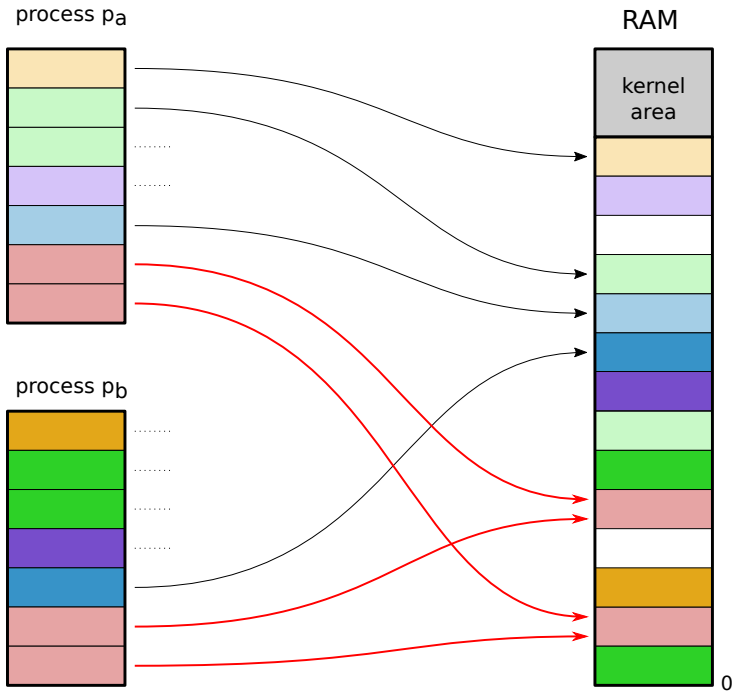

Figura 18.3: Compartilhamento de páginas.

Áreas de memória compartilhada também podem ser usadas para permitir a comunicação entre processos. Para tal, dois ou mais processos solicitam ao núcleo o mapeamento de uma área de memória sobre a qual ambos podem ler e escrever. Como os endereços lógicos acessados nessa área serão mapeados sobre a mesma área de memória física, o que cada processo escrever nessa área poderá ser lido pelos demais, imediatamente. A Seção 9.3 traz informações mais detalhadas sobre a comunicação entre processos através de memória compartilhada.

18.2 Copy-on-write (COW)

O mecanismo de compartilhamento de memória não é restrito apenas às seções de código. Em princípio, toda área de memória protegida contra escrita pode ser compartilhada, o que pode incluir áreas de dados constantes, como tabelas de constantes, textos de ajuda, etc., proporcionando ainda mais economia de memória.

A técnica conhecida como *copy-on-write* (CoW, ou *copiar ao escrever*), é uma estratégia usada quando o núcleo do SO precisa copiar páginas de um espaço de memória para outro. Ela pode ser aplicada, por exemplo, quando um processo cria outro através da chamada de sistema `fork()` (vide Seção 5.3). Nessa chamada, deve ser feita uma cópia do conteúdo de memória do processo pai para o processo filho.

Na técnica CoW, ao invés de copiar as páginas do processo pai para o filho, o núcleo compartilha as páginas e as marca como "somente leitura" e "copy-on-write", usando os flags das tabelas de página (Seção 14.7.2). Quando um dos processos tentar escrever em uma dessas páginas, a proteção contra escrita irá provocar uma falta de página, ativando o núcleo. Este então irá verificar que se trata de uma página "copy on write", criará uma cópia separada daquela página para o processo que deseja fazer a escrita, e removerá a proteção contra escrita dessa cópia.

Os principais passos dessa estratégia estão ilustrados na Figura 18.4 e detalhados a seguir:

1. Dois processos têm páginas compartilhadas pelo mecanismo CoW; as páginas somente podem ser acessadas em leitura;
2. o processo p_a tenta escrever em uma página compartilhada;
3. a MMU nega o acesso e provoca uma falta de página, ativando o núcleo do SO;
4. o núcleo faz uma cópia da página em outro quadro de memória RAM;
5. o núcleo ajusta a tabela de páginas de p_a para apontar para a cópia da página, limpa o flag *CoW* e permite a escrita na cópia;
6. o processo p_a continua a executar e consegue completar sua operação de escrita;
7. p_b continua a acessar o conteúdo original da página.

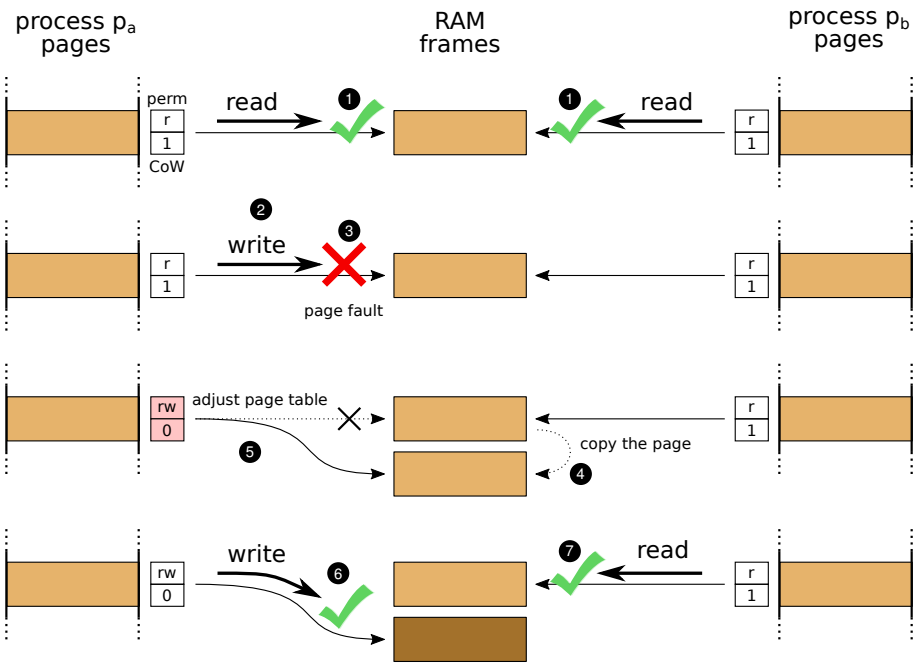

Figura 18.4: Estratégia de compartilhamento com *Copy-on-Write*.

Todo esse procedimento é feito de forma transparente para os processos envolvidos, visando compartilhar ao máximo as áreas de memória dos processos e assim otimizar o uso da RAM. Esse mecanismo é mais efetivo em sistemas baseados em páginas, porque normalmente as páginas são menores que os segmentos. A maioria dos sistemas operacionais atuais (Linux, Windows, Solaris, FreeBSD, etc.) usa esse mecanismo.

18.3 Mapeamento de arquivo em memória

Uma funcionalidade importante dos sistemas operacionais atuais é o mapeamento de arquivos em memória, que surgiu no SunOS 4.0 [Vahalia, 1996]. Esse mapeamento consiste em associar uma área específica de memória do processo a um arquivo em disco: cada byte no arquivo corresponderá então a um byte naquela área de memória, sequencialmente. A Figura 18.5 ilustra esse conceito.

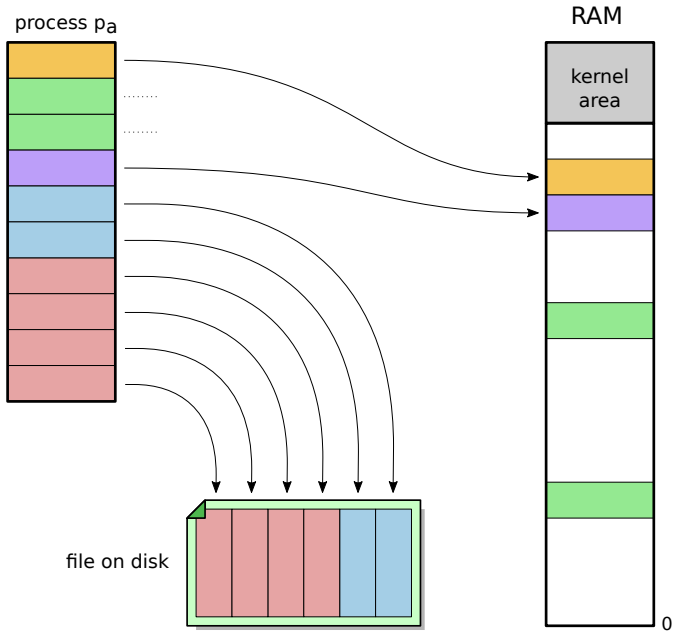

Figura 18.5: Mapeamento de arquivo em memória.

O conteúdo do arquivo mapeado pode ser acessado pelo processo através de leitura e escrita de bytes naquela área de memória que o mapeia. É importante observar que o mapeamento não implica em carregar o arquivo inteiro na memória. Ao invés disso, é criada uma área de memória com o mesmo tamanho do arquivo e suas páginas são ajustadas para "acesso proibido", usando os flags da tabela de páginas (Seção 14.7.2). Quando o processo tentar acessar uma posição de memória naquela área, a MMU irá gerar uma falta da página para o núcleo, que irá então buscar e carregar naquela página o conteúdo correspondente do arquivo mapeado. Esse procedimento é denominado *paginação sob demanda* (*demand paging*).

Os principais passos da paginação sob demanda estão ilustrados na Figura 18.6 e detalhados a seguir:

1. Um arquivo é mapeado em uma área de memória do processo; as páginas dessa área são marcadas como inacessíveis;
2. o processo p_a tenta acessar uma página da área mapeada;
3. a MMU nega o acesso e gera uma falta de página, ativando o núcleo do SO;
4. o núcleo carrega o conteúdo correspondente do arquivo na memória;
5. o núcleo ajusta a tabela de páginas de p_a;

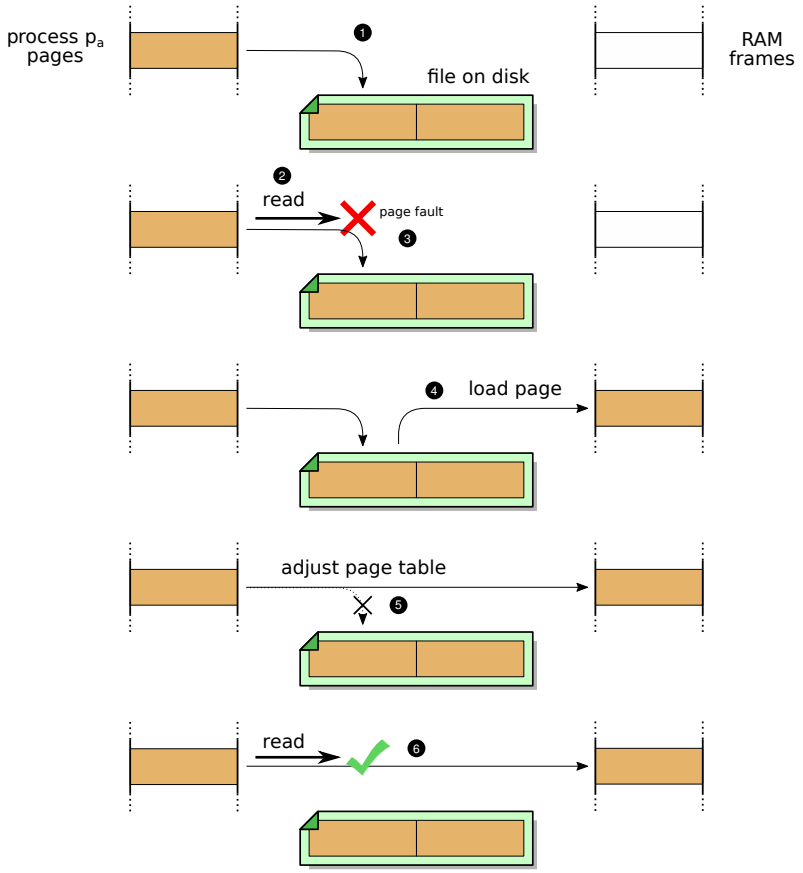

Figura 18.6: Paginação sob demanda.

6. o processo p_a continua a executar e consegue completar seu acesso.

Em relação às operações de escrita, os mapeamentos podem ser *compartilhados* ou *privados*. Em um mapeamento compartilhado, as escritas feitas pelo processo na área mapeada são copiadas no arquivo, para que possam ser vistas por outros processos que abrirem aquele arquivo. No mapeamento privado, é usada a técnica *copy on write* para não propagar as escritas ao arquivo mapeado, que permanece intacto.

A paginação sob demanda é particularmente interessante para o lançamento de processos. Um arquivo executável é estruturado em seções, que contêm código, dados inicializados, etc., refletindo as seções de memória de um processo (Seção 15.2). A carga do código de um novo processo pode então ser feita pelo simples mapeamento das seções correspondentes do programa executável na memória do processo. À medida em que o processo executar, ele irá gerar faltas de páginas e provocar a carga das páginas correspondentes do programa executável. Com isso, somente os trechos de código efetivamente executados serão carregados na memória, o que é particularmente útil em programas grandes.

Outra vantagem do mapeamento de arquivos em memória é o compartilhamento de código executável. Caso mais processos que executem o mesmo programa sejam lançados, estes poderão usar as páginas já carregadas em memória pelo primeiro processo, gerando economia de memória e rapidez de carga.

Referências

U. Vahalia. *UNIX Internals – The New Frontiers*. Prentice-Hall, 1996.