SISTEMAS OPERACIONAIS

CAPÍTULO 3 - SE VÁRIOS PROCESSOS SÃO EXECUTADOS E SE COMUNICAM AO MESMO TEMPO, COMO PODEMOS CONTROLÁ-LOS?

Osvaldo de Souza

Introdução

Se o SO está ativo, haverá processos em execução. Ou seja, enquanto estiver ligado, o microcomputador nunca está inativo, pois a CPU sempre está ativa.
Com a possibilidade de novas arquiteturas, os microcomputadores são dotados não apenas de uma, mas duas ou mais CPUs que operam conjuntamente, sob o controle de uma delas.
Significa que, em tais arquiteturas, mais de um processo é executado em um determinado momento. Aplicativos podem ser construídos para tirar vantagens desse tipo de estrutura, fazendo com que seja extraído o máximo de capacidade da CPU, minimizando, assim, o tempo total de processamento que venha a ser necessário.
Seja em microcomputadores com várias ou apenas uma CPU, o cenário é o mesmo: vários processos ativos e disputando a CPU, a memória e todos os outros tipos de recursos disponíveis.
Diante disso, surgem questões de como fazer o controle e a comunicação entre os processos e o SO e entre os próprios processos.
Em geral, a comunicação entre os processos e o SO se dá por meio das APIs, abstrações para as chamadas de sistemas providas pelo sistema operacional, para intermediar o acesso ao *hardware*, ou então para acesso a funcionalidades de *software* que o próprio SO oferece. Então como as comunicações acontecem? Como são controladas? Sob o intenso tráfego de dados entre a memória RAM, o disco e a CPU, quem gerencia esse volume de comunicações?
Você terá respostas para essas perguntas e mais outras informações com a leitura deste capítulo.
Vamos lá? Bons estudos!

3.1 A comunicação de mensagens e a sincronização entre os processos

Percebe-se, facilmente, que os processos, enquanto executados, geram uma intensa comunicação interna, portanto, produzem uma enorme quantidade de mensagens que são trocadas entre eles, na forma das APIs, chamadas de sistema e até mesmo recursos compartilhados. Tal é a importância dessa intensa comunicação que qualquer erro leva a trágicas consequências, como o encerramento prematuro de processos e até falhas gerais do SO.
A comunicação ocorre e é necessária para que os processos utilizem os recursos que estão disponíveis no microcomputador por meio do sistema operacional. Mas você precisa compreender que as transferências de dados entre a unidade de processamento central, a memória RAM e o disco rígido também são tipos de comunicação.
Você pode se perguntar: como assim? Por exemplo, quando você edita uma imagem em seu microcomputador usando um aplicativo gráfico, para que a imagem seja exibida no *display*, foram necessárias diversas comunicações, que, inclusive, envolveram a transferência da imagem original para a memória RAM e também para a memória do *display*.
Portanto, a comunicação é caracterizada por mensagens curtas, médias e grandes. Ainda em nosso exemplo, considere que durante e após a edição da imagem, você precisa salvar o que foi feito. Esse processo de registrar o trabalho no disco rígido envolve um conjunto extenso de comunicações.

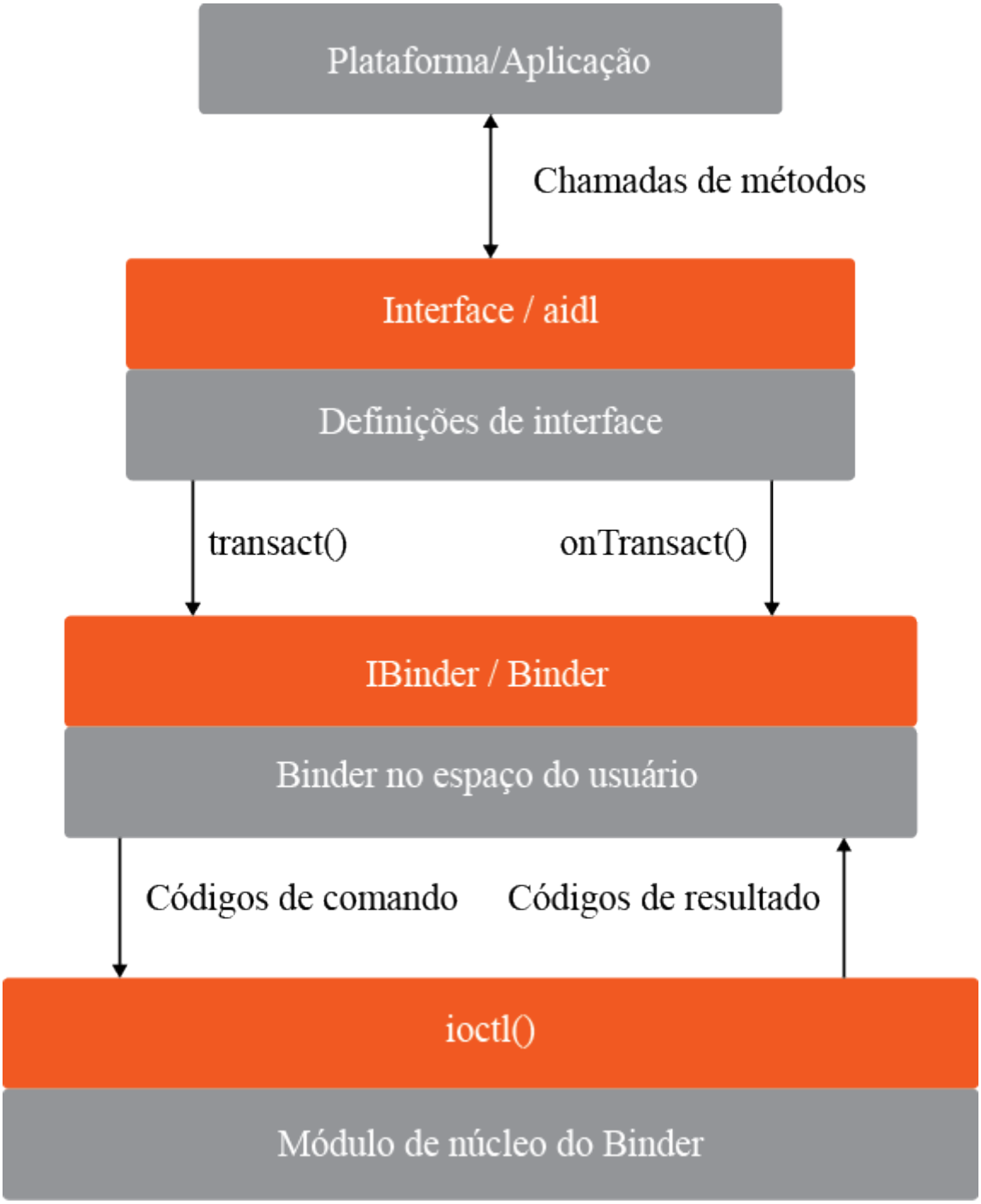

Figura 1 - O encadeamento da comunicação no Binder do SO Android permite que um aplicativo do espaço do usuário (no topo da figura) acesse o espaço do núcleo.
Fonte: TANENBAUM; BOS, 2016, p. 565.

A exemplo do entrelaçamento das comunicações entre os processos, observe a Figura anterior que apresenta a comunicação que ocorre (generalizada) para o uso do Binder no sistema operacional *Android* (TANENBAUM; BOS, 2016). O Binder é utilizado para que aplicações do espaço do usuário possam invocar e utilizar recursos que estejam disponíveis em equipamentos alcançados por intermédio da rede de dados. Trata-se, portanto, do uso de recursos compartilhados a distância. E esse compartilhamento requer o entrelaçamento das comunicações entre processos feito de forma correta. Além disso, devemos considerar os problemas relacionados à memória RAM, tais como: proteção, alocação e desalocação de trechos, localização e fragmentação das partes livres da memória.

Ou seja, tudo precisa ocorrer em sintonia entre os processos e os diversos *hardwares* que compõem o microcomputador, o que é trabalho do sistema operacional.

A comunicação entre os processos é estabelecida de uma maneira fortemente estruturada, normalmente chamada de protocolo. O protocolo pode ser muito simples, do tipo "quando um estiver falando, o outro apenas ouve", mas há situações em que requer vários dados repassados do emissor para o receptor. Esses dados repassados são chamados de conteúdo da comunicação ou parâmetros.

VOCÊ SABIA?

Embora a Internet seja bastante popular, a ideia de construir repositórios nos quais as pessoas possam ter acesso e interagirem com dados ou programas é muito antiga. Esses repositórios antigos eram chamados de BBS, do inglês *bulletin board system*, e foi a partir deles que provavelmente nasceu a ideia da WWW. O funcionamento dos BBS era totalmente dependente dos protocolos, que estabeleciam como você podia enviar uma mensagem ou um arquivo. Diferente de hoje, havia limites de *download*, então quando você acessava um BBS, tinha (nos bons casos) direito de baixar 300 *kilobytes* de dados, usando uma conexão de *modem* que transmitia, no máximo, 300 *bytes* por segundo. Outro mundo, não é mesmo?

Tanenbaum e Bos (2016, p. 28) trazem que "processos relacionados que estão cooperando para finalizar alguma tarefa muitas vezes precisam comunicar-se entre si e sincronizar as atividades". Isso estabelece uma das razões pelas quais os processos se comunicam. Outras razões incluem: solicitar algum tipo de serviço; a sincronização de alguma tarefa; o envio ou recebimento de dados; a notificação de resultado de processamento; e o erro ou sucesso nas tarefas. Portanto, como há diferentes motivos pelos quais os processos se comunicam, há também diferentes maneiras de estabelecer e realizar a comunicação. Mesmo assim, há previsibilidade quanto aos possíveis tipos de comunicação e dessa forma torna-se possível criar protocolos e/ou tipos padrões de mensagens que serão trocadas.

Suponha que exista um processo X que forneça os serviços de data e hora. Para atender uma demanda entre X e os outros processos que venham a se comunicar com ele, cria-se o seguinte padrão para a comunicação: para solicitar a data deve ser usada a mensagem *obterData*() e para obter a hora, a mensagem *obterHora*(). Como você percebeu, essas mensagens são parecidas com as APIs e o modo de usá-las estabelece o protocolo de comunicação. Suponha ainda que seja possível enviar uma mensagem para alterar a data e/ou a hora: *definirData* (valor) e *definirHora*(valor). O valor que surge dentro dos parênteses é um indicativo de que que junto à mensagem deve ser enviada nova data ou hora para a qual X deve alterar os valores anteriores. Assim, para alterar a data, por exemplo, para 22 de maio de 2018, usaríamos: *definirData*("22/05/2018").

Essa comunicação é denominada comunicação entre processos (IPC), do inglês *interprocess communication.* Tanenbaum e Bos (2016) estabelecem três questões que precisam ser tratadas em relação ao funcionamento dos processos e que envolvem IPC na solução.

A primeira diz respeito à simples necessidade de troca de mensagens (visando troca de dados ou serviços). Quando processos desejam apenas se comunicarem, basta o uso do mecanismo correto para o envio do tipo certo de mensagem.

A segunda questão refere-se ao estabelecimento da sequência correta de ações quando dependências estão presentes. Isso é para evitar que processos se atrapalhem, o que geralmente acontece quando dois ou mais estão interessados no mesmo recurso, o que chamamos de relação produtor-consumidor. Como exemplo, imagine o processo A que lê e carrega dados do disco rígido; o processo B que decodifica dados relativos a músicas e vídeos; e o processo C que apresenta músicas e vídeos para o usuário. Essa relação precisa funcionar da seguinte

maneira: A(B(C())), isto é, o processo A utiliza os dados produzidos por B, que utiliza os dados produzidos por C. Isso estabelece uma sequência correta de ações visando o alcance de um resultado.
Já a terceira questão refere-se a processos não se atrapalharem mutuamente. Isso significa que o interesse em um mesmo recurso por vários processos, como o estabelecimento da necessidade de compartilhamento de algo. Processos de uma mesma família podem desejar acessar um mesmo arquivo de dados, ou uma mesma região da memória. Quando uma situação como essa ocorre, encontramos uma condição de corrida, o que torna o trecho do processo envolvido uma região crítica.

VOCÊ QUER LER?

A comunicação entre os processos não é o único tipo de comunicação realizada pelos microcomputadores. Há também aquela realizada entre os microcomputadores, inclusive com diferentes sistemas operacionais.

Essa comunicação é tão intensa que requer uma enorme infraestrutura de dados. Você conhece alguma rede desse tipo? Ou já usou alguma rede assim? A internet é um exemplo. Para compreender esse universo que transcende a comunicação *interprocess*, recomendamos a leitura do livro "Comunicação de Dados e Rede de Computadores" (FOROUZAN, 2010).

Para a solução de acesso a um recurso compartilhado é necessário estabelecer uma política que defina o momento certo para um processo acessar o recurso e, ao mesmo tempo, definir que os demais não podem ter o acesso.
Isso requer algum tipo de solução que garanta a exclusividade de acesso à região crítica do código que trata do recurso compartilhado. O que é importante, pois, do contrário, pode ocorrer inconsistência nos dados, já que há uma ordem específica em que os processos deveriam acessar e modificar os dados do recurso compartilhado. Essa solução em particular é conhecida por exclusão mútua. Tanenbaum e Bos (2016) propõem o estabelecimento de quatro condições para que se encontre uma boa solução para a exclusão mútua:

1. Dois processos jamais podem estar simultaneamente dentro de duas regiões críticas;
2. Nenhuma suposição pode ser feita a respeito de velocidades ou do número de CPUs;
3. Nenhum processo executando fora de sua região crítica pode bloquear qualquer recurso;
4. Nenhum processo deve ser obrigado a esperar eternamente para entrar em sua região crítica (TANENBAUM; BOS, 2016, p. 83).

A Figura a seguir ilustra a exclusão mútua na visão dos autores.

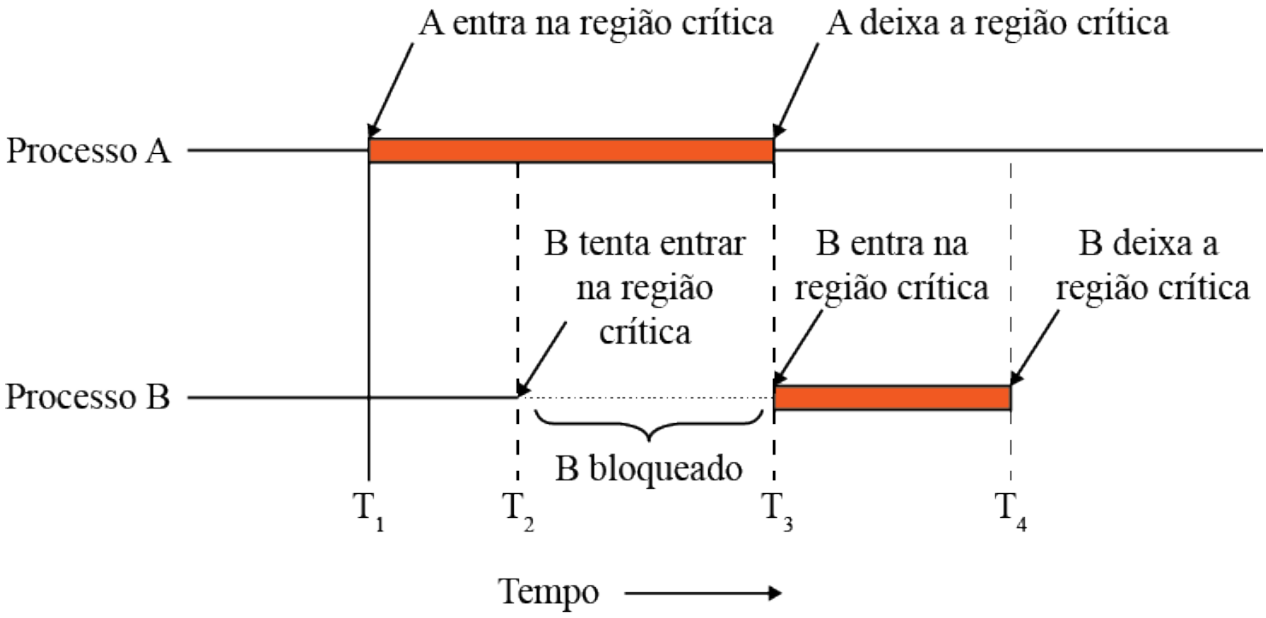

Figura 2 - Visualização de como pode ser estabelecida a exclusão mútua pelo tratamento adequado à região crítica.
Fonte: TANENBAUM; BOS, 2016, p. 83.

Podemos compreender o problema da exclusão mútua das regiões críticas como uma questão de sincronismo entre os processos (MELO NETO, 2014). Perceba que o bloqueio dos demais processos estabelece uma proteção para que dois ou mais não estejam na região crítica, porém não trata da questão da ordem de execução. Dependendo de como a exclusão mútua é implementada, pode ocorrer de um processo nunca conseguir chegar à região crítica, pois ela não garante uma ordem de execução, assim quem chegou em segundo pode ser preterido por outros processos.

3.1.1 Algoritmo de Peterson/semáforos

Então são dois grandes problemas que precisam ser resolvidos: 1 - garantir que dois processos não cheguem à região crítica simultaneamente; 2 - garantir que todos os processos que compartilham a região crítica tenham a sua vez de acesso.

VOCÊ QUER LER?

Um jeito divertido para compreender alguns algoritmos importantes e também certos conceitos que envolvem programação, sistemas operacionais, decisões etc. pode ser encontrado no livro “Algoritmos para viver: A ciência exata das decisões humanas” (CHRISTIAN; GRIFFITHS, 2017). No texto, os autores traçam paralelos entre a computação (e os algoritmos) e a vida humana. Se tiver curiosidade, parte do livro pode ser obtido gratuitamente na internet.

Uma proposta para solucionar os dois problemas foi dada por Peterson (TANENBAUM; BOS, 2016), que é do tipo “espera ocupada”, cujo algoritmo na linguagem de programação C pode ser visto no Quadro a seguir.

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

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

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

    other = 1 – process;          /* o posto 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 – espera ocupada*/
}
void leave_region (int process)         /* processo: quem está saindo */
{
   interested[process]=FALSE;         /* indica a saída da região critica*/
}
```

Quadro 1 - Solução de Peterson é um espera ocupada para a exclusão mútua e necessita do conhecimento prévio da quantidade de processos que compartilham o recurso.
Fonte: TANENBAUM; BOS, 2016, p. 85.

Na solução apresentada no Quadro, o grande segredo está na condição *while (turn == process && interested [other]== TRUE*. Podemos ler o trecho da seguinte forma: permaneça esperando enquanto for a nossa vez e o outro processo ainda está interessado. Observe que o uso de *leave_region* indica que o valor de *interested* para o processo na região crítica é informar que não tem mais interesse: *interested[process]=FALSE*. Quando isso ocorrer, o processo que estava no "permaneça enquanto (...)" perceberá que quem o estava usando perdeu o interesse, logo podemos usar.

Contra a solução proposta por Peterson está o fato de ela necessitar do conhecimento prévio da quantidade de processos que irão compartilhar o recurso. Caso a solução seja empregada estratégia que permita o aumento do número de processos que compartilham o recurso, a solução não funcionará. Pesa também contra essa solução, o fato de consistir em uma espera ocupada, obviamente implicando no consumo da CPU e significativo desperdício da capacidade de processamento.

VOCÊ O CONHECE?

Richard Matthew Stallman, conhecido pelo apelido RMS, tem grande participação na computação moderna no que diz respeito aos sistemas operacionais e à linguagem C. Stallman está à frente da Fundação de *Software* GNU e é um dos responsáveis pela popularização do *software* livre, dos quais o *Linux* é o mais conhecido. Há também o compilador da linguagem C, aliás, toda uma família de compiladores para diversas linguagens intensamente usadas.

Ainda sobre a solução de Peterson, observe que existem algumas variáveis (*turn, interested*) usadas para regular o acesso à região crítica. Essas variáveis atuam como semáforos e têm a função de sinalizar para que os demais processos percebam se podem ou não acessar à região crítica. Portanto, temos uma técnica combinada de semáforos e inserção obrigatória de código de controle da região crítica.

VOCÊ QUER VER?

As regiões críticas evitam situações em que o compartilhamento de recursos pode acarretar dano ao SO ou perda de dados. Um caso muito particular de problema que pode acontecer quando recursos são compartilhados (ou disputados) por diferentes processos resulta no travamento de ao menos dois processos. Estamos falando do travamento mortal (*deadlock*), mas há um lugar em que o *deadlock* pode ser usado para salvar sua vida. Isso aconteceu em *Tron – uma odisséia eletrônica* (LISBERGER, 1982). No filme, durante frenéticas disputas (e fugas) em motocicletas virtuais, o único jeito de vencer é bloqueando o adversário, isto é, criando um *deadlock* que o leva a destruição.

A inserção obrigatória de código de controle de região crítica refere-se à necessidade de todos os processos incorporarem as chamadas *enter_region* e *leave_region*. Sem essa obediência a estratégia falharia.

A problemática que envolve o controle das regiões críticas pode ser aumentada quando consideramos um cenário em que não apenas diferentes processos pertençam a mesma família, mas, inclusive, a famílias diferentes. Considere que um usuário inicia um aplicativo de navegador para acessar diversas páginas de conteúdo na *web*. Suponha agora que o usuário também inicie um aplicativo para acessar um serviço de banco e, ao mesmo tempo, resolva ouvir música em algum aplicativo diretamente da internet.

Todos esses aplicativos estão fazendo uso da comunicação pela rede de dados, ou seja, todos competem pelo uso da rede, recurso controlado pelo SO. Como isso acontece? Na verdade, em uma visão generalista, são problemas resolvidos com o mesmo tipo de estratégia: permitir que apenas um processo use o recurso por vez.

No caso do envio e recebimento de dados pela rede, a grande saída é delegar toda a atividade a duas únicas partes do sistema operacional. Todas as aplicações e seus inúmeros processos sequer tentam fazer acesso direto ao recurso compartilhado (rede), eles enviam e recebem dados por meio dessas partes específicas do sistema operacional, fornecido por uma chamada de sistema.

Centralizando o acesso ao recurso e garantindo que apenas um único caminho esteja disponível para o uso do recurso, a ocorrência de condições de corrida torna-se mínima. Essa estratégia também resolve o problema de garantir que nenhum processo fique esperando para sempre sem ser atendido.
Por isso, basta que as solicitações sejam colocadas em uma fila e atendidas com algum algoritmo simples, que pode ser do tipo FIFO, uma política simplificada de lidar com um conjunto de requisições de recursos. Não se preocupe que essa política será explicada um pouco adiante do capítulo. Enquanto isso, vamos seguir nosso estudo com o tema gerenciamento de memória real.

3.2 Gerenciamento de memória real

Memória real refere-se à memória de acesso randômico (RAM), do inglês *random access memory*. É aquela na qual os processos são efetivamente armazenados em tempo de execução. Se um microcomputador possui 8 *gigabytes* de memória, significa que esse é o limite máximo de memória que pode ser usado na máquina, sem o uso de virtualização. Eventualmente, o proprietário pode comprar mais memória RAM e adicioná-la ao *hardware* , respeitando os limites aceitos pela placa principal instalada no microcomputador.
A memória RAM normalmente tem o *design* de uma pequena placa de circuito impresso com uma região de conectores que fica em contato na placa principal, nos *memory slots*.

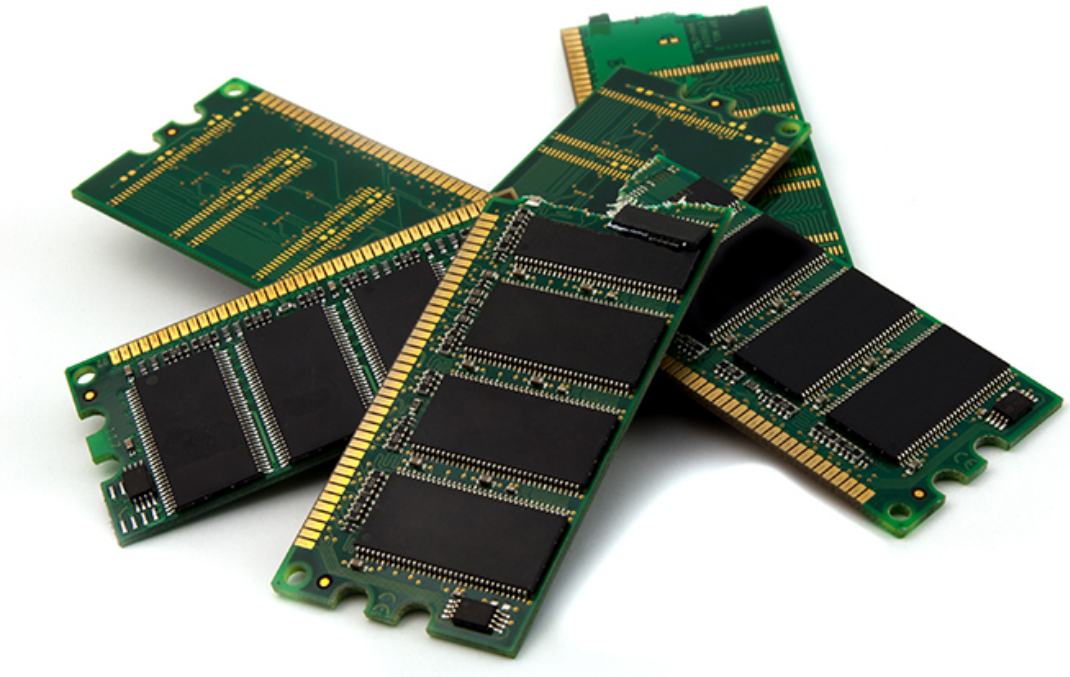

Figura 3 - Exemplos de cartões de memória, componentes altamente especializados e que facilitam a expansão de memória real em um microcomputador.
Fonte: Aigars Reinholds, Shutterstock, 2018.

A memória real é acessada pela CPU e também pela *direct memory access* (DMA) e em ambas as modalidades o SO pode exercer controle. A tecnologia DMA foi criada para que a transferência ocorresse diretamente por seus canais de comunicação e não exigisse a participação da CPU.
Mas essa tecnologia nem sempre esteve presente. Os primeiros microcomputadores não dispunham da DMA e todo o tráfego de dados gerado pelos processos, e que envolviam a RAM e o disco rígido, contavam com o trabalho da CPU. Nessa abordagem antiga, a CPU também tinha a função de processar a transferência de dados e quando um processo precisava movimentar quantidades significativas de dados, ocorria uma sobrecarga no sistema.
Ainda hoje, mesmo em microcomputadores modernos, alguns tipos de transferência de dados que envolvem apenas a memória RAM são manipulados exclusivamente pela CPU, impactando no desempenho do sistema como um todo.

3.2.1 Conceitos básicos e funcionalidades

Os processos não fazem, de fato, acesso direto ao endereçamento real da memória. É produzido um mapeamento com blocos, denominado espaço de endereçamento, cuja finalidade é facilitar o gerenciamento da memória.
Por meio do espaço de endereçamento é criada uma abstração que faz com que o processo tenha "sua" própria memória. Assim, o endereçamento de um processo sempre será diferente do utilizado por outro, exceto quando há memória compartilhada entre eles.
O gerenciador de memória fornece uma parcela da memória real para o processo quando solicitado. Caso a quantidade de memória pedida não esteja disponível, o processo é informado, e então é fornecida a quantidade máxima possível. Nesse caso, cabe ao processo decidir se irá realizar o processamento esperado ou se ele deve ser interrompido devido à falta de recursos. A partir desse pressuposto, torna-se necessário que os programadores também se esforcem para que seus programas sejam adaptativos, de forma que essa adaptação diz respeito aos seus programas terem capacidade para realizar o processamento necessário de modo fragmentado, sem que essa fragmentação comprometa o todo. Não estamos aqui nos referindo à uma divisão do trabalho feita em processos menores, o que caracterizaria a programação paralela, mas sim de um processo que precisa, por exemplo, realizar uma transformação em uma imagem. Esse processo deve ser projetado de maneira que se possa trabalhar na imagem inteira por vez (caso tenha conseguido muita memória RAM), ou, então, que se consiga aplicar a transformação da imagem por partes, até que o trabalho tenha sido todo processado e transformado.
Há situações em que não é possível fragmentar o processamento, restando ao processo abortar a missão, notificando o usuário que ocorreu um erro devido à falta de memória.

3.2.2 Monoprogramação sem trocas de processos ou Paginação

No início do desenvolvimento dos microcomputadores e das primeiras versões de SO, o *hardware* não possuía características de proteção da memória. A proteção, no caso, refere-se a impedir que um determinado processo enderece e manipule memória fora do seu espaço de endereçamento.
Um microcomputador sem *hardware* de proteção torna a multiprogramação impossível. Qualquer programa pode apresentar mau funcionamento acidental (ou intencional) e comprometer toda a memória do computador, destruindo o SO e todos os dados em RAM.
Hoje os microcomputadores modernos contam com a presença de *hardware* de proteção, mas ainda existe um conjunto enorme de equipamentos, conhecidos como equipamentos embarcados, que não possuem proteção de memória, sendo então monoprogramados.
E como isso é feito? Bom, na monoprogramação não ocorre a necessidade de trocas de processos e, por consequência direta, não ocorre remanejamento da memória, não existindo também a necessidade de paginação. O trabalho de gerenciamento da memória é praticamente desnecessário, visto que toda a memória fica à disposição do processo que esteja em execução.

3.3 Gerenciamento de memória

Nos cenários da multiprogramação, a alocação da memória é dinâmica. A dinamicidade vem da imprevisibilidade da quantidade de processos que podem ser executados e também da quantidade de memória requisitada por esses processos.

Um processo pode solicitar determinada quantidade de memória e, momentos depois, aumentar ou diminuir esse pedido. A variabilidade na alocação da memória pelos processos gera um enorme trabalho para o gerenciador e traz problemas relacionados à criação de espaços livres entre os espaços alocados na memória.

Suponha que exista um conjunto de aplicativos do usuário, representados pelos processos (A, B, C e D), que nas linhas de tempo ilustradas na Figura a seguir (a, b, c, d, e, f, g), podem ou não estar em execução nas respectivas linhas. Perceba a dinâmica entre a ocupação e liberação da memória com o término e reinício de alguns desses aplicativos ao longo do tempo.

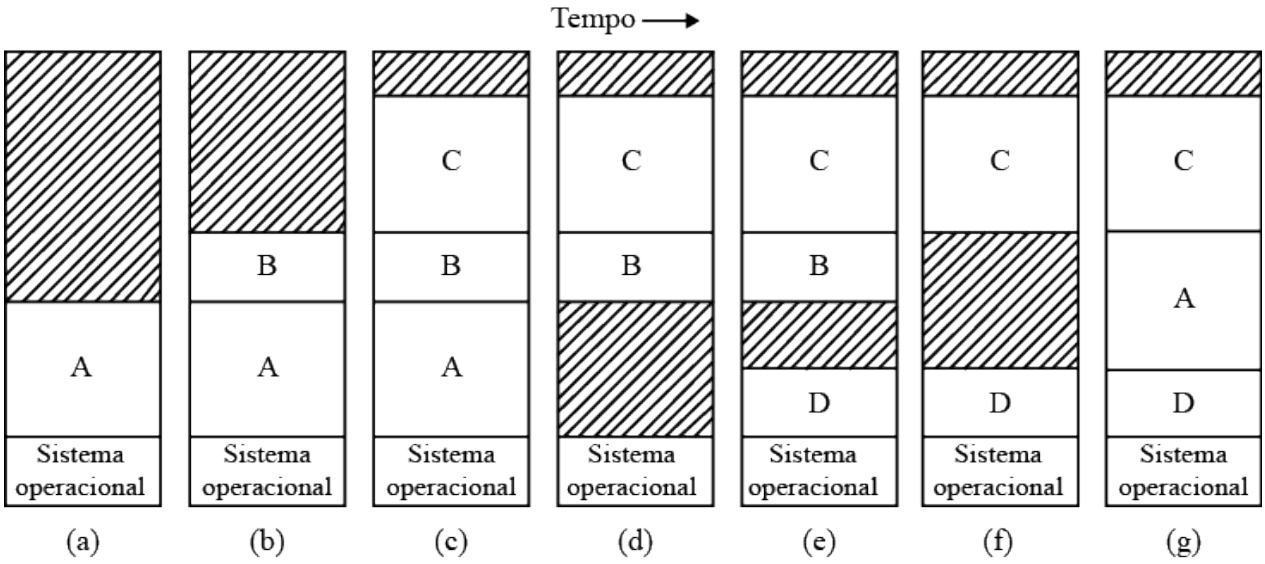

Figura 4 - Ilustração com a alocação de memória em sete momentos distintos (a – g) e os espaços não alocados entre os que estejam comprometidos com determinados processos.
Fonte: TANENBAUM; BOS, 2016, p. 130.

Note que na Figura, os espaços hachurados indicam trechos que não estão alocados. Em certas situações, um processo pode ter um pedido de memória negado, mesmo que haja livre, bastando apenas que não exista um bloco de memória livre que seja contíguo ao bloco de memória do processo.

VOCÊ QUER VER?

Você já deve ter visto notícia de um carro controlado por um sistema operacional, mas é pouco provável que conheça um sistema operacional que possa atuar como organizador pessoal, ajudando as pessoas a completarem suas tarefas e até dando conselhos amorosos. Bem, ainda não chegamos nesse estágio na vida real, mas essa discussão foi retratada nos cinemas. No filme *Ela* (JONZE, 2013), o ator Joaquim Phoenix interpreta um homem solitário que começa um relacionamento amoroso com um sistema operacional e nos faz pensar até que ponto os SOs podem chegar.

Para minimizar esse tipo de problema, a memória é organizada pelo SO em pequenas frações. Se essas frações forem muito pequenas, haverá um aumento da complexidade do gerenciamento. Portanto, é preciso procurar um equilíbrio entre a quantidade e o tamanho das frações.
Quando a memória é fracionada, para ter a organização e o gerenciamento facilitados, é necessário algum mecanismo para o controle, como o mapa de *bits.*

3.3.1 Mapa de bits

O mapa de *bits* é uma maneira de endereçar (com objetivo de controlar) pequenas partes da memória. O controle funciona assim: elege-se uma quantidade de memória que será representada por um *bit*. Divide-se a quantidade de memória total pela quantidade que foi eleita. O resultado será a quantidade de *bits* necessários no mapa.

(a) Uma parte da memória com cinco processos e três espaços. As marcas indicam as unidades de alocação de memória. As regiões sobreadas (0 no mapa de bits) estão livres. (b) Mapa de bits correspondente. (c) A mesma informação como lista.

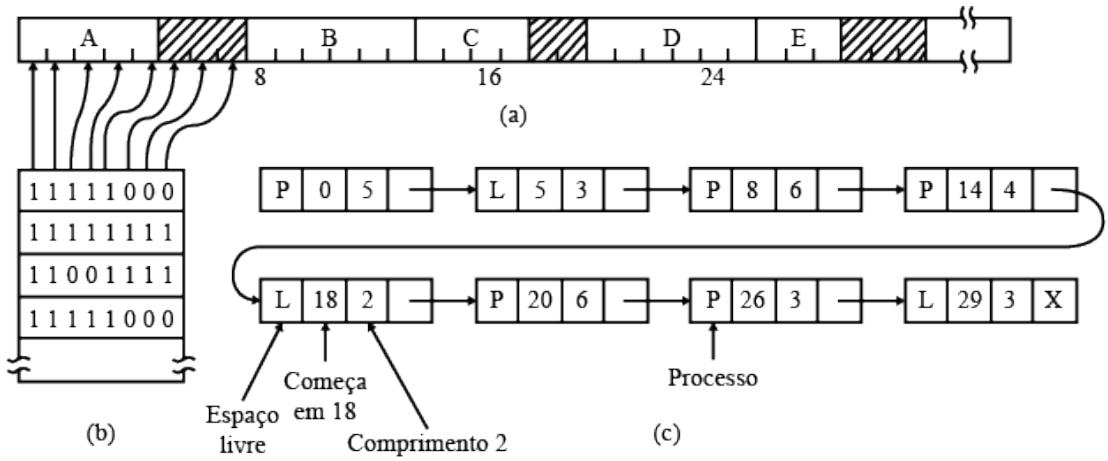

Figura 5 - Exemplo de mapa de bits, o uso do mapa simplifica o controle sobre uma parte da memória estar ou não alocada.
Fonte: TANENBAUM; BOS, 2016, p. 132.

Na Figura anterior, a memória está representada na parte (a), o mapa de *bits* em (b), e os processos com uso da memória em (c). Olhando da esquerda para a direita, perceba que os números "uns" (1) referem-se às porções alocadas da memória e os zeros (0) às porções livres.
Observando o lado superior esquerdo da parte (c), podemos perceber o P05, no qual "P" indica que a região está alocada a um processo, "0" que o trecho alocado tem início em zero e o 5 que o trecho tem o tamanho de 5 *bytes*, e assim sucessivamente.

3.3.2 Alocação de segmentos livres

Na abordagem baseada em mapa de *bits*, o maior problema consiste na alocação de segmentos livres. Quando um processo é carregado para a memória é necessário encontrar um trecho grande o suficiente para que possa contê-lo. Esse trabalho requer que o gerenciador de memória percorra todo o mapa de *bits* para encontrar a região livre que satisfaça o tamanho necessário. É uma operação lenta, pois, inevitavelmente, trata-se de uma busca sequencial.
O tempo necessário para isso é diretamente proporcional à quantidade de processos que estejam em memória. Isso ocorre, pois quanto mais processos houver, maior será a necessidade de comparações nos trechos livres encontrados. E lembrando-se de que ao final, pode-se chegar à conclusão de que não há memória contigua livre, mesmo em situações em que, ao somar os vários trechos de memória livre, o resultado seja muito maior do que se procurava inicialmente.
Considere que ocorra uma falha no gerenciamento e na alocação de segmentos livres, o resultado seria uma catástrofe no sistema operacional, pois um determinado processo poderia sobrescrever os dados de outro processo, destruindo-o ou modificando-o de forma que ele realize processamentos diferentes daqueles para os quais tenha sido construído.
Essa preocupação é real e sistemas operacionais com falhas na alocação e proteção de memória estão expostos a vulnerabilidades que abrem espaço para *hackers* (programadores experientes que usam das falhas para acesso a funcionalidades ou dados no sistema operacional) instalaram trechos de código mal-intencionado e prejudicar os interesses do usuário. Muito bem, agora que apresentamos um panorama sobre a memória real, vamos para outro tipo: a virtual.

3.4 Gerenciamento de memória virtual

O gerenciamento de memória virtual tem por finalidade permitir que os microcomputadores pareçam ter muito mais memória do que realmente a instalada no *hardware*. Junto com essa funcionalidade acaba sendo necessário implementar uma série de recursos para o controle dessa abstração.
Lembre-se sempre de que os processos precisam estar na memória RAM quando forem executados. Caso um processo (ou parte dele) esteja em disco rígido (na memória virtual), será necessário mover o trecho do disco rígido para a memória RAM. Obviamente isso requer tempo: de processamento da CPU, dos canais de DMA e do disco rígido.

VOCÊ SABIA?

Por vezes nos deparamos com aquela situação em que o microcomputador está aparentemente "travado", mas há uma intensa atividade no disco rígido, percebida pelo som do acionamento constante ou pela luz (*led*) no microcomputador. Essa situação costuma ocorrer quando há um número muito grande de processos em execução ou que exista um consumo muito grande de memória. O sistema operacional então precisa constantemente fazer trocas entre a memória RAM e o disco rígido. Esse volume de troca, sendo alto, pode levar o sistema a um travamento de fato, principalmente se o algoritmo de escalonamento que estiver em uso seja por simples fatia de tempo (*quantum*). Portanto, em microcomputadores com pouca memória é sempre melhor manter o mínimo de aplicativos em execução para se obter um melhor desempenho no geral.

Um microcomputador com quantidade adequada de memória RAM pode ter seu desempenho melhorado com o uso de memória virtual, mas esse resultado pode não ser tão positivo se a quantidade de memória real for pequena e os aplicativos que o usuário iniciar requisitarem muita memória.

Suponha existir um microcomputador com 2 *gigabytes* de memória livre durante a execução por um determinado SO. São carregados 3 processos, e cada um requer 1 *gigabyte* de memória. Utilizando-se os 2 *gigabytes* livres, seria possível manter dois processos em memória, porém como são três, um terá de ser continuamente carregado do disco rígido para a RAM e depois carregado da RAM para o disco rígido.

Se consideramos que o tempo de transferência de dados para um disco rígido é muito maior do que o tempo de transferência para a memória RAM, é fácil perceber que este microcomputador estará constantemente efetuando leitura e escrita no disco rígido, o que irá afetar o desempenho geral da máquina.

CASO

Um microcomputador será utilizado em atividades que irão requerer uma intensa movimentação de dados entre a memória de acesso randômico e o disco rígido. Um exemplo em que isso acontece é nas chamadas ilhas de edição. Essas ilhas são um ou mais equipamentos destinados à manipulação de fluxos de vídeos e/ou grandes imagens (de alta resolução). Nessas ilhas é muito comum fazer edições em trechos de vídeos, acrescentando trilha sonora, efeitos visuais etc. Para esse tipo de trabalho, além de muita memória RAM, é importante que se tenha um disco rígido de alto desempenho, o qual deve ser medido pela taxa de transferência de leitura e escrita. Quando maior for essa taxa, melhor será o desempenho da ilha. O contrário também vale.

Mas como minimizar essa situação? Como transformar um processo de 1 *gigabyte* em um menor? Aliás, isso é possível?

Bem, segundo Silberschatz e Galvin (2015), um processo de 1 *gigabyte* não pode ser transformado em um menor, mas podemos dividi-lo em partes menores. Estatisticamente, pode-se afirmar que um processo (de 1 *gigabyte*) não é executado de uma só vez na CPU, apenas uma parte dele, em cada uma das vezes que o processo estiver no estado de executando.
Assim, surge uma estratégia de fracionamento da memória que será utilizada pelos processos em partes menores e diretamente mapeada na memória real.
Para poder implementar a memória virtual, a memória real do microcomputador é organizada em uma abstração denominada de páginas, compostas por endereços contíguos. O endereçamento das páginas não é correspondente aos endereços reais da memória do microcomputador.

3.4.1 Conceitos de Paginação

Quando um processo é criado e iniciado, ele pode ser grande demais para ser mantido inteiro na memória. Isso pode ocorrer por não haver memória livre suficiente, ou devido ao total existente no microcomputador ser menor do que o necessário para o processo. Além disso, quando o processo é executado pela primeira vez (ou em qualquer uma das vezes que estiver em execução), ele faz suas requisições de memória e esta pode ser maior do que a memória real disponível no momento. Caso seja o cenário, o SO irá prover mais memória para o processo utilizar (ou até mesmo para acomodar o próprio processo), lançando mão da memória virtual.
As páginas de um processo são simplesmente blocos de divisões da memória relativos ao processo e podem estar na memória real ou armazenadas no disco rígido.
Quando o processo faz referência (indica que vai usar) a um bloco (página) que não está na memória principal (seja memória para todo o processo ou parte dele), é necessário trazer esse bloco do disco rígido para a memória RAM. E para isso acontecer, algum bloco deve sair. Nessa situação, surge uma pergunta crucial ao gerenciamento e implementação da paginação: qual bloco deve sair?

3.4.2 Implementação de Paginação

Umas das estratégias para gerir a troca de páginas na memória é o algoritmo *first in, first out* (FIFO). Mesmo sendo uma estratégia simples de escolha, é muito utilizada na gestão de filas de pedidos de recursos. Tanenbaun e Bos nos mostram (2016) que:

> o FIFO remove a página menos recentemente carregada, independente de quando essa página foi referenciada pela última vez. Há um contador associado a cada quadro de página e, de início, todos os contadores estão definidos como 0. Após cada falta de página ter sido tratada, o contador de cada página que se encontra na memória no momento é aumentado em (1) um (TANENBAUM; BOS, 2016, p. 352).

Na estratégia FIFO, a solução é a mais simples possível: quem chegou primeiro, agora tem de sair.
É como em um brinquedo de parque de diversões, a roda gigante, por exemplo. A roda vai girando e todos vão se divertindo. Perceba que as pessoas que estão no brinquedo não chegaram ao mesmo tempo (é o mesmo que ocorre com as páginas de memória virtual que estão na memória real). Quando chega uma nova pessoa para subir na roda gigante, alguém tem de sair. Sairá (como todos nós sabemos) quem estava lá por mais tempo. O primeiro que havia chegado na roda gigante, será o primeiro a sair. E isso é o que define o funcionamento do algoritmo FIFO.
Se no sistema operacional estiver em uso uma política de escalonamento por *quantum* de tempo, esse sobre e desce da roda gigante, isto é, entra e sai da memória RAM, ocorrerá com frequência. Pois, quando o processo que está em execução já finalizou seu *quantum* de tempo, ele sairá da memória e da CPU, e um novo processo será carregado. O novo processo certamente terá suas próprias páginas de memória, o que é completamente normal.

Tanenbaum e Bos (2016) nos dizem que o conjunto de páginas do processo é denominado conjunto de trabalho e pode ter páginas mais e menos utilizadas.

Os autores apresentam um mapeamento possível entre as páginas virtuais e a memória RAM, dividido, por exemplo, em oito quadros de página, como pode ser visto na Figura a seguir.

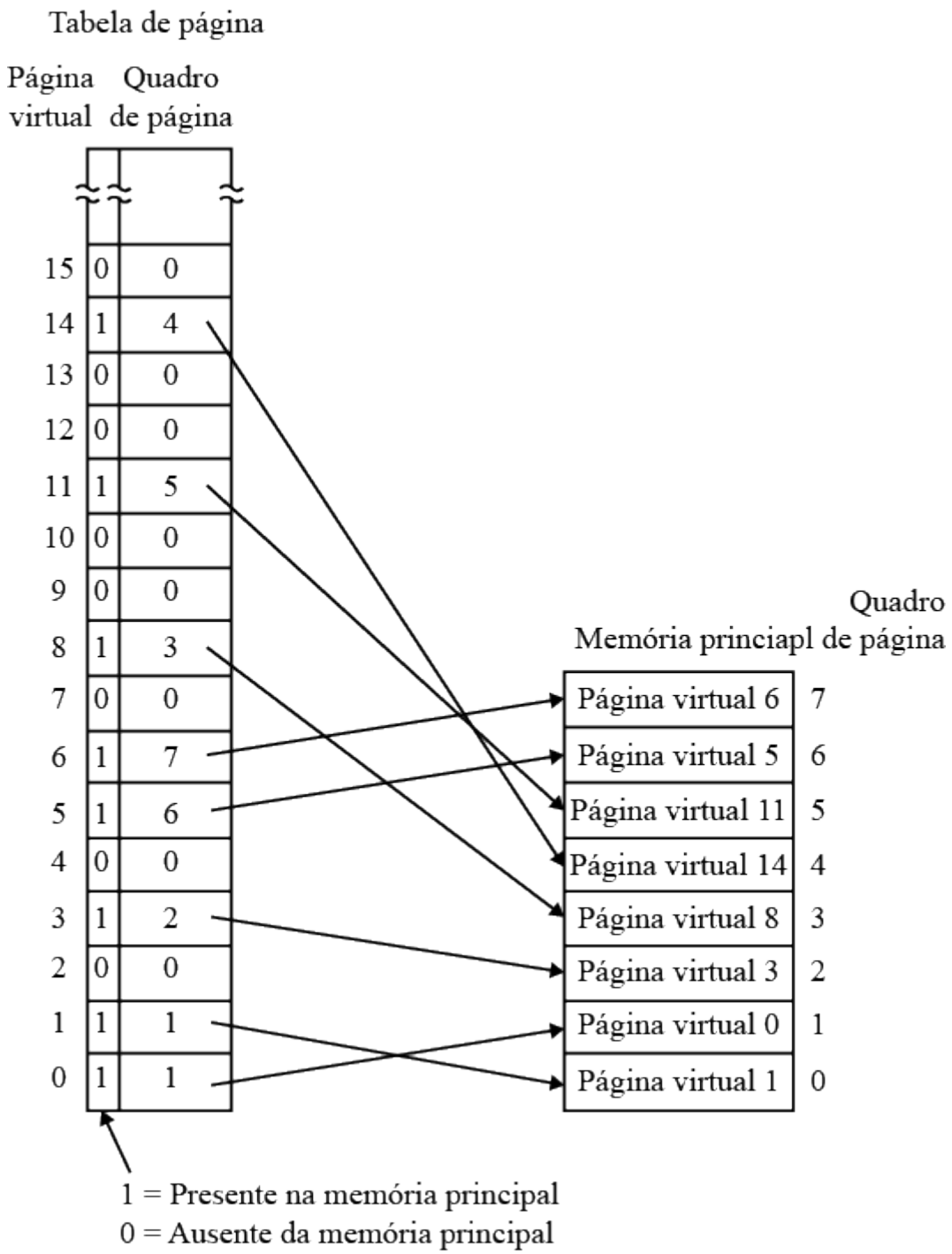

Figura 6 - Mapeamento entre 16 páginas de memória virtual em 8 páginas da memória principal do microcomputador.

Fonte: TANENBAUM; BOS, 2016, p. 350.

O processo de troca de página pelo algoritmo FIFO ocorre sempre que uma das dezesseis páginas da memória virtual for referenciada e não estiver em nenhum dos oito quadros nos quais a memória principal foi dividida.

Se considerarmos também o cenário em que ocorre a programação paralela, haverá um potencial alto de troca de páginas na memória (BARCELLOS, 2002).
Na ocorrência de um volume grande de troca de páginas, pode haver perda de desempenho de modo geral por limitar as funções de entrada e saída da memória à velocidade do disco rígido, que é significativamente menor do que o da memória RAM. Portanto, em programação paralela, é importante que o processo seja executado em um microcomputador corretamente dimensionado para a tarefa.

Síntese

Compreendemos aqui, a dinâmica da comunicação de dados que ocorre por meio de mensagens trocadas entre os processos. Vimos também como se dá o controle da memória do microcomputador. Destacamos que sem a comunicação (e o seu controle) os processos não poderiam compartilhar recursos ou comunicarem-se efetivamente com o sistema operacional. Vimos também como se dá o controle da memória, compreendemos a importância da memória virtual e de como ela permite que o microcomputador vá além dos seus limites de memória.
Neste capítulo, você teve a oportunidade de:

- compreender situações em que o uso compartilhado de recursos se torna uma grande dificuldade de gerenciamento;
- entender que a comunicação entre os processos está no centro da problemática que envolve o funcionamento do sistema operacional;
- compreender como a comunicação entre os processos é crucial para que possa haver cooperação entre diferentes processos e que essa cooperação só pode ser garantida com o estabelecimento de políticas rígidas de comunicação;
- examinar os cenários que permeiam o uso da memória e da grande limitação que é a baixa disponibilidade de memória RAM para um microcomputador;
- entender que um microcomputador pode executar mais processos do que a sua real memória permite ao usar a memória virtual;
- compreender os conceitos da paginação e de como ela permite o mapeamento entre a memória virtual e a memória real, cuja dinâmica é administrada pelo sistema operacional.

Bibliografia

BARCELLOS, M. P. Programação Paralela e Distribuída em Java. In: 2ª Escola Regional de Alto Desempenho - ERAD 2002. São Leopoldo. **Anais...** São Leopoldo, p. 181-192, 2002. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erad-rs/2002/007.pdf>. Acesso em: 31/05/2018.
CHRISTIAN, B.; GRIFITHS, T. **Algoritmos para viver:** a ciência exata das decisões humanas. São Paulo: Companhia das Letras, 2017.
DEITEL, H. **Sistemas Operacionais.** 3. ed. São. Paulo: Pearson, 2005.
_____; DEITEL, P. J. **C: Como programar**. 6. ed. São Paulo: Pearson Prentice Hall, 2011.
FOROUZAN, B. A. **Comunicação de Dados e Rede de Computadores.** 4. ed. Porto Alegre: McGrawHill, 2010.
JONZE, S. **Ela**. Direção: Spike Jonze. Produção: Megan Ellison; Spike Jonze; Vincent Landay. EUA, 2013.
LISBERGER, S. **Tron – Uma Odisséia Eletrônica**. Direção: Steven Lisberger. Produção: Donald Kushner. EUA, 1982.

MACHADO, F. B.; MAIA, L. P. **Arquitetura de Sistemas Operacionais**. 5. ed. Rio de Janeiro: LTC. 2011. Disponível na Biblioteca Virtual Laureate: <https://laureatebrasil.blackboard.com/webapps/blackboard/content/listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset>. Acesso em: 24/05/2018.

MELO NETO, A. **Estrutura dos Sistemas Operacionais.** USP. 2014. Disponível em: <https://www.ime.usp.br/~adao/teso.pdf>. Acesso em: 17/05/2018.

SILBERSCHATZ, A.; GALVIN, P. B. **Fundamentos de Sistemas Operacionais**. 9. ed. São Paulo, LTC 2015. Disponível na Biblioteca Virtual Laureate: <https://laureatebrasil.blackboard.com/webapps/blackboard/content/listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset>. Acesso em: 24/05/2018.

TANENBAUM, A. S.; BOS, H. **Sistemas Operacionais Modernos**. 4. ed. São Paulo: Person Education do Brasil, 2016. Disponível na Biblioteca Virtual Laureate: <https://laureatebrasil.blackboard.com/webapps/blackboard/content/listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset>. Acesso em: 24/05/2018.