Cap. 1: Introdução

Se todo programador de aplicativos tivesse de compreender como todas as partes do computador funcionam em detalhe, nenhum código jamais seria escrito. Por essa razão, computadores são equipados com um dispositivo de software chamado de sistema operacional.

O programa com o qual os usuários interagem, normalmente chamado de shell (ou interpretador de comandos) quando ele é baseado em texto e de GUI (Graphical User Interface) quando ele usa ícones, na realidade não é parte do sistema operacional, embora use esse sistema para realizar o seu trabalho. A maioria dos computadores tem dois modos de operação: modo núcleo/kernel (modo supervisor) e modo usuário. No modo núcleo, há acesso completo a todo o hardware para a execução de qualquer instrução que a máquina for capaz de executar (Figura 1.1).

Uma distinção importante entre o sistema operacional e o software normal (modo usuário) é que se um usuário não gosta de um leitor de e-mail em particular, ele é livre para conseguir um leitor diferente ou escrever o seu próprio, se assim quiser; ele não é livre para escrever seu próprio tratador de interrupção de relógio, o qual faz parte do sistema operacional e é protegido por hardware contra tentativas dos usuários de modificá-lo.

O código-fonte do coração de um sistema operacional como Linux ou Windows tem cerca de cinco milhões de linhas. Para entender o que isso significa, considere como seria imprimir cinco milhões de linhas em forma de livro, com 50 linhas por página e 1.000 páginas por volume. Seriam necessários 100 volumes para listar um sistema operacional desse tamanho — em essência, uma estante de livros inteira. Imagine-se conseguindo um trabalho de manutenção de um sistema operacional e no primeiro dia seu chefe o leva até uma estante de livros com o código e diz: “Você precisa aprender isso”. E isso é apenas para a parte que opera no núcleo. Quando bibliotecas compartilhadas essenciais são incluídas, o Windows tem bem mais de 70 milhões de linhas de código ou 10 a 20 estantes de livros. E isso exclui softwares de aplicação básicos (do tipo Windows Explorer, Windows Media Player e outros). Deve estar claro agora por que sistemas operacionais têm uma longa vida — eles são dificílimos de escrever, e tendo escrito um, o proprietário reluta em jogá-lo fora e começar de novo.

1.1 O que é um sistema operacional?

Os sistemas operacionais realizam duas funções essencialmente não relacionadas: fornecer a programadores de aplicativos (e programas aplicativos, claro) um conjunto de recursos abstratos limpo em vez de recursos confusos de hardware, e gerenciar esses recursos de hardware.

1.1.1 O sistema operacional como uma máquina estendida

Considere os discos rígidos modernos SATA (Serial ATA) usados na maioria dos computadores. Um livro (ANDERSON, 2007) descrevendo uma versão inicial da interface do disco — o que um programador deveria saber para usar o disco —, tinha mais de 450 páginas. Desde então, a interface foi revista múltiplas vezes e é mais complicada do que em 2007. É claro que nenhum programador são iria querer lidar com esse disco em nível de hardware. Em vez disso, um software, chamado driver de disco, lida com o hardware e fornece uma interface para ler e escrever blocos de dados, sem entrar nos detalhes. Sistemas operacionais contêm muitos drivers para controlar dispositivos de E/S. Mas mesmo esse nível é baixo demais para a maioria dos aplicativos. Por essa razão, todos os sistemas operacionais fornecem mais um nível de abstração para se utilizarem discos: arquivos. Usando essa abstração, os programas podem criar, escrever e ler arquivos, sem ter de lidar com os detalhes complexos de como o hardware realmente funciona.

Essa abstração é a chave para gerenciar toda essa complexidade. Boas abstrações transformam uma tarefa praticamente impossível em duas tarefas gerenciáveis. A primeira é definir e implementar as abstrações. A segunda é utilizá-las para solucionar o problema à mão.

A função dos sistemas operacionais é criar boas abstrações e então implementar e gerenciar os objetos abstratos criados desse modo. Neste livro, falaremos muito sobre abstrações. Elas são uma das chaves para compreendermos os sistemas operacionais. Uma das principais tarefas dos sistemas operacionais é esconder o hardware e em vez disso apresentar programas (e seus programadores) com abstrações de qualidade, limpas, elegantes e consistentes com as quais trabalhar. Sistemas operacionais transformam o feio em belo.

Um usuário de Linux executando Gnome ou KDE vê uma interface muito diferente daquela vista por um usuário Linux trabalhando diretamente sobre o X Window System, mas as abstrações do sistema operacional subjacente são as mesmas em ambos os casos. Neste livro, esmiuçaremos o estudo das abstrações fornecidas aos programas aplicativos, mas falaremos bem menos sobre interfaces com o usuário.

1.1.2 O sistema operacional como um gerenciador de recursos

O conceito de um sistema operacional como fundamentalmente fornecendo abstrações para programas aplicativos é uma visão top-down (abstração de cima para baixo). Uma visão alternativa, bottom-up (abstração de baixo para cima), sustenta que o sistema operacional está ali para gerenciar todas as partes de um sistema complexo.

Computadores modernos consistem de processadores, memórias, temporizadores, discos, dispositivos apontadores do tipo mouse, interfaces de rede, impressoras e uma ampla gama de outros dispositivos. Na visão bottom-up, a função do sistema operacional é fornecer uma alocação ordenada e controlada dos processadores, memórias e dispositivos de E/S entre os vários programas competindo por eles. Sistemas operacionais modernos permitem que múltiplos programas estejam na memória e sejam executados ao mesmo tempo. Imagine o que aconteceria se três programas executados em um determinado computador tentassem todos imprimir sua saída simultaneamente na mesma impressora. O sistema operacional pode trazer ordem para o caos em potencial armazenando temporariamente no disco, toda a saída destinada para a impressora.

A principal função do sistema operacional é manter um controle sobre quais programas estão usando qual recurso, conceder recursos requisitados, contabilizar o seu uso, assim como mediar requisições conflitantes de diferentes programas e usuários.

O gerenciamento de recursos inclui a multiplexação (compartilhamento) de recursos de duas maneiras diferentes: no tempo e no espaço. Quando um recurso é multiplexado no tempo, diferentes programas ou usuários se revezam usando-o. Por exemplo, com apenas uma CPU e múltiplos programas querendo ser executados nela, o sistema operacional primeiro aloca a CPU para um programa, então, após ele ter sido executado por tempo suficiente, outro programa passa a fazer uso da CPU. Determinar como o recurso é multiplexado no tempo — quem vai em seguida e por quanto tempo — é a tarefa do sistema operacional. Outro exemplo da multiplexação no tempo é o compartilhamento da impressora. Quando múltiplas saídas de impressão estão na fila para serem impressas em uma única impressora, uma decisão tem de ser tomada sobre qual deve ser impressa em seguida. O outro tipo é a multiplexação de espaço. Em vez de os clientes se revezarem, cada um tem direito a uma parte do recurso. Por exemplo, a memória principal é normalmente dividida entre vários programas sendo executados, de modo que cada um pode ser residente ao mesmo tempo (por exemplo, a fim de se revezar usando a CPU). Presumindo que há memória suficiente para manter múltiplos programas, é mais eficiente manter vários programas na memória ao mesmo tempo do que dar a um deles toda ela, especialmente se o programa precisa apenas de uma pequena fração do total. É claro, isso gera questões de justiça, proteção e assim por diante, e cabe ao sistema operacional solucioná-las. Outro recurso que é multiplexado no espaço é o disco. Em muitos sistemas um único disco pode conter arquivos de muitos usuários ao mesmo tempo. Alocar espaço de disco e controlar quem está usando quais blocos do disco é uma tarefa típica do sistema operacional.

1.2 Conceitos de sistemas operacionais

Sistemas operacionais fornecem determinados conceitos e abstrações básicos, como processos, espaços de endereços e arquivos, que são fundamentais para compreendê-los.

1.2.1 Processos

Processo é um programa em execução. Associado a cada processo está o espaço de endereçamento, uma lista de posições de memória que vai de 0 a algum máximo, onde o processo pode ler e escrever. O espaço de endereçamento contém o programa executável, os dados do programa e sua pilha. Associado a cada processo há um conjunto de recursos, abrangendo registradores (incluindo o contador de programa e o ponteiro de pilha), uma lista de arquivos abertos, alarmes pendentes, listas de processos relacionados e demais informações necessárias para executar um programa. Um processo é na essência um contêiner que armazena todas as informações necessárias para executá-lo.

Num sistema de multiprogramação, por exemplo, poderíamos ter três processos ativos: o editor de vídeo, o navegador da web e o receptor de e-mail. Periodicamente, o sistema operacional decide parar de executar um processo e começa a executar outro, talvez porque o primeiro utilizou mais do que sua parcela de tempo da CPU no último segundo ou dois.

Quando um processo é suspenso temporariamente, ele deve ser reiniciado mais tarde no exato estado em que estava quando foi parado. Assim, todas as informações a respeito do processo são salvos em algum lugar durante a suspensão. O processo pode ter vários arquivos abertos para leitura ao mesmo tempo. Há um ponteiro associado a cada um desses arquivos dando a posição atual (isto é, o número do byte ou registro a ser lido em seguida). Quando um processo está temporariamente suspenso, todos esses ponteiros têm de ser salvos de maneira que uma chamada read executada após o processo ter sido reiniciado vá ler os dados corretos.

As informações a respeito de cada processo, fora o conteúdo do seu próprio espaço de endereçamento, estão armazenadas em uma tabela do sistema operacional chamada de tabela de processos, que é um arranjo de estruturas, uma para cada processo existente no momento. Desse modo, um processo (suspenso) consiste em seu espaço de endereçamento, em geral chamado de imagem do núcleo (em homenagem às memórias de núcleo magnético usadas antigamente), e de sua entrada na tabela de processo, que armazena os conteúdos de seus registradores e muitos outros itens necessários para reiniciar o processo mais tarde.

As principais chamadas de sistema de gerenciamento de processos são as que lidam com sua criação e término. Quando o processo tiver terminado a compilação, ele executa uma chamada de sistema para se autofinalizar. Se um processo pode criar um ou mais processos (chamados de processos filhos), e estes por sua vez podem criar processos filhos, chegamos logo à estrutura da árvore de processo. Vale ressaltar que o Windows não tem uma hierarquia de processo como o UNIX, então não há um conceito de um processo pai e um processo filho. Após um processo ser criado, o criador e criatura são iguais.

Processos relacionados que estão cooperando para finalizar alguma tarefa muitas vezes precisam comunicar-se entre si e sincronizar as atividades. Essa comunicação é chamada de comunicação entre processos. Outras chamadas de sistemas de processos permitem requisitar mais memória (ou liberar memória não utilizada), esperar que um processo filho termine e sobrepor seu programa por um diferente. Um processo que está se comunicando com outro em um computador diferente envia mensagens para o processo remoto por intermédio de uma rede de computadores. Para evitar a perda de uma mensagem/resposta, o emissor pode pedir para o seu próprio sistema operacional notificá-lo após um tempo, de maneira que ele possa retransmitir a mensagem se nenhuma confirmação tiver sido recebida ainda. Após ligar esse temporizador, o programa pode continuar executando outra tarefa. Decorrido o tempo definido, o sistema operacional envia um sinal para o processo. O sinal faz que o processo suspenda o que ele esteja fazendo, salve seus registradores na pilha e comece a executar uma rotina para tratamento desse sinal, por exemplo,

para retransmitir uma mensagem presumivelmente perdida. Quando a rotina de tratamento desse sinal encerra sua ação, o processo em execução é reiniciado no estado em que se encontrava um instante antes do sinal.

Sinais são os análogos em software das interrupções em hardwares e podem ser gerados por uma série de causas além de temporizadores expirando. A cada pessoa autorizada a usar um sistema é designada uma UID (User IDentification — identificação do usuário) pelo administrador do sistema. Todo processo iniciado tem a UID da pessoa que o iniciou. Um processo filho tem a mesma UID que o seu processo pai. Usuários podem ser membros de grupos, cada qual com uma GID (Group IDentification — identificação do grupo). Uma UID, chamada de superusuário (em UNIX), ou Administrador (no Windows), tem um poder especial e pode passar por cima de muitas das regras de proteção.

1.2.2 Espaços de endereçamento

Todo computador tem alguma memória principal que ele usa para armazenar programas em execução. Em um sistema operacional muito simples, apenas um programa de cada vez está na memória. Para executar um segundo programa, o primeiro tem de ser removido.

Sistemas operacionais mais sofisticados permitem que múltiplos programas estejam na memória ao mesmo tempo. Para evitar que interfiram entre si (e com o sistema operacional), algum tipo de mecanismo de proteção é necessário. Embora esse mecanismo deva estar no hardware, ele é controlado pelo sistema operacional. Este último ponto de vista diz respeito ao gerenciamento e à proteção da memória principal do computador. Uma questão diferente relacionada à memória, mas igualmente importante, é o gerenciamento de espaços de endereçamento dos processos.

No caso mais simples, a quantidade máxima de espaço de endereços que um processo tem é menor do que a memória principal. Dessa maneira, um processo pode preencher todo o seu espaço de endereçamento e haverá espaço suficiente na memória principal para armazená-lo inteiramente. No entanto, em muitos computadores os endereços são de 32 ou 64 bits, dando um espaço de endereçamento de 2^{32} e 2^{64} , respectivamente. O que acontece se um processo tem mais espaço de endereçamento do que o computador tem de memória principal e o processo quer usá-lo inteiramente? Nos primeiros computadores, ele não teria sorte. Hoje, existe uma técnica chamada memória virtual, como já mencionado, na qual o sistema operacional mantém parte do espaço de endereçamento na memória principal e parte no disco, enviando trechos entre eles para lá e para cá conforme a necessidade. Na essência, o sistema operacional cria a abstração de um espaço de endereçamento como o conjunto de endereços ao qual um processo pode se referir. O gerenciamento de espaços de endereçamento e da memória física forma uma parte importante do que faz um sistema operacional.

1.2.3 Arquivos

Uma função importante do sistema operacional é esconder as peculiaridades dos discos e outros dispositivos de E/S e apresentar ao programador um modelo agradável e claro de arquivos que sejam independentes dos dispositivos. Chamadas de sistema são necessárias para criar, remover, ler e escrever arquivos. Antes que um arquivo possa ser lido, ele deve ser localizado no disco e aberto, e após ter sido lido, deve ser fechado.

O conceito de diretório é utilizado pelos sistemas operacionais para manter os arquivos agrupados. Chamadas de sistema também são necessárias para criar e remover diretórios. Entradas de diretório podem ser de arquivos ou de outros diretórios. Esse modelo também dá origem a uma hierarquia — o sistema de arquivos.

Ambas as hierarquias de processos e arquivos são organizadas como árvores, mas a similaridade para aí. Hierarquias de processos em geral não são muito profundas (mais do que três níveis é incomum), enquanto hierarquias de arquivos costumam ter quatro, cinco, ou mesmo mais

níveis de profundidade. Hierarquias de processos tipicamente têm vida curta, em geral minutos no máximo, enquanto hierarquias de diretórios podem existir por anos.

Propriedade e proteção também diferem para processos e arquivos. Normalmente, apenas um processo pai pode controlar ou mesmo acessar um processo filho, mas quase sempre existem mecanismos para permitir que arquivos e diretórios sejam lidos por um grupo mais amplo do que apenas o proprietário.

Todo arquivo dentro de uma hierarquia de diretório pode ser especificado fornecendo o seu nome de caminho a partir do topo da hierarquia do diretório, o diretório-raiz. Esses nomes de caminho absolutos consistem na lista de diretórios que precisam ser percorridos a partir do diretório-raiz para se chegar ao arquivo, com barras separando os componentes.

Antes que um arquivo possa ser lido ou escrito, ele precisa ser aberto, momento em que as permissões são conferidas. Se o acesso for permitido, o sistema retorna um pequeno valor inteiro, chamado descritor de arquivo, para usá-lo em operações subsequentes. Se o acesso for proibido, um código de erro é retornado.

Outro conceito importante em UNIX é o de montagem do sistema de arquivos. A maioria dos computadores têm portas USB ou discos rígidos externos que podem ser conectados. Para fornecer uma maneira elegante de lidar com essa mídia removível, a UNIX permite que o sistema de arquivos no pendrive ou flash drive seja agregado à árvore principal, através do sistema de arquivos-raiz sempre que seja pedido pelo programa.

Outro conceito importante em UNIX é o arquivo especial, que permitem que dispositivos de E/S se pareçam com arquivos. Eles podem ser lidos e escritos com as mesmas chamadas de sistema que são usadas para ler e escrever arquivos. Há 2 tipos especiais: arquivos especiais de bloco e arquivos especiais de caracteres. Arquivos especiais de bloco são usados para modelar dispositivos que consistem em uma coleção de blocos aleatoriamente endereçáveis, como discos. Ao abrir um arquivo especial de bloco e ler, digamos, bloco 4, um programa pode acessar diretamente o quarto bloco no dispositivo, sem levar em consideração a estrutura do sistema de arquivo contido nele. De modo similar, arquivos especiais de caracteres são usados para modelar impressoras, modems e outros dispositivos que aceitam ou enviam um fluxo de caracteres. Por convenção, os arquivos especiais são mantidos no diretório /dev. Por exemplo, /dev/lp pode ser a impressora (line printer).

Há ainda o conceito de pipe. Uma espécie de pseudoarquivo usado para conectar dois processos ou arquivos. Quando o processo A quer enviar dados para o processo B, ele escreve no pipe como se ele fosse um arquivo de saída. O processo B pode ler os dados a partir do pipe como se ele fosse um arquivo de entrada. A comunicação entre os processos em UNIX se parece muito com a leitura e escrita de arquivos comuns.

1.2.4 Entrada/Saída

Todos os computadores têm dispositivos físicos para obter entradas e produzir saídas. Existem muitos tipos de dispositivos de entrada e de saída, incluindo teclados, monitores, impressoras e assim por diante. Cabe ao sistema operacional gerenciá-los.

Todo sistema operacional tem um subsistema de E/S para gerenciar os dispositivos de E/S. Alguns softwares de E/S são independentes do dispositivo, isto é, aplicam-se igualmente bem a muitos ou a todos dispositivos de E/S. Outras partes dele, como drivers de dispositivo, são específicos a dispositivos de E/S particulares.

1.2.5 Proteção

Cabe ao sistema operacional gerenciar a segurança do sistema de maneira que os arquivos, por exemplo, sejam acessíveis somente por usuários autorizados. Arquivos em UNIX são protegidos designando-se a cada arquivo um código de proteção binário de 9 bits. O código de proteção consiste de três campos de 3 bits, um para o proprietário, um para os outros membros do grupo do proprietário (usuários são divididos em grupos pelo administrador do sistema) e um para todos os

demais usuários. Cada campo tem um bit de permissão de leitura, um bit de permissão de escrita e um bit de permissão de execução. Esses 3 bits são conhecidos como os bits rwx. Por exemplo, o código de proteção rwxr-x--x significa que o proprietário pode ler (read), escrever (write), ou executar (execute) o arquivo, que outros membros do grupo podem ler ou executar (mas não escrever) o arquivo e que todos os demais podem executar (mas não ler ou escrever) o arquivo. Para um diretório, x indica permissão de busca. Um traço significa que a permissão correspondente está ausente.

Além da proteção ao arquivo, há muitas outras ques-tões de segurança. Proteger o sistema de intrusos indesejados, humanos ou não (por exemplo, vírus) é uma delas.

1.2.6 O interpretador de comandos (shell)

O sistema operacional é o código que executa as chamadas de sistema. Editores, compiladores, montadores, ligadores (linkers), programas utilitários e interpretadores de comandos não fazem parte do sistema operacional.

Embora não faça parte do sistema operacional, o interpretador de comandos UNIX (shell), por exemplo, faz um uso intensivo de muitos aspectos do sistema operacional e serve assim como um bom exemplo de como as chamadas de sistema são usadas. Ele é a principal interface entre um usuário e o sistema operacional, a não ser que o usuário esteja usando uma interface gráfica. Muitos shells existem, incluindo, sh, csh, ksh e bash. Todos eles dão suporte à funcionalidade descrita a seguir, derivada do shell (sh) original. O shell inicia-se como um terminal de entrada e saída-padrão, emitindo um caractere de prompt, como o cifrão do dólar, que diz ao usuário que o shell está esperando para aceitar um comando. Se o usuário agora digitar *date,* por exemplo, o shell cria um processo filho e executa o programa date como um filho. Enquanto o processo filho estiver em execução, o shell espera que ele termine. Quando o filho termina, o shell emite o sinal de prompt de novo e tenta ler a próxima linha de entrada. O usuário pode especificar que a saída-padrão seja redirecionada para um arquivo, por exemplo, *date >file.* De modo similar, a entrada-padrão pode ser redirecionada, como em *sort <file1 >file2* que invoca o programa sort com a entrada vindo de file1 e a saída enviada para file2. A saída de um programa pode ser usada como entrada por outro programa conectando-os por meio de um pipe. Assim, *cat file1 file2 file3 | sort >/dev/lp* invoca o programa cat para concatenar três arquivos e enviar a saída para que o sort organize todas as linhas em ordem alfabética. A saída de sort é redirecionada para o arquivo /dev/lp, tipicamente a impressora. Se um usuário coloca um & após um comando, o shell não espera que ele termine. Em vez disso, ele dá um prompt imediatamente. Em consequência, cat file1 file2 file3 | sort >/dev/lp & inicia o sort como uma tarefa de segundo plano, permitindo que o usuário continue trabalhando normalmente enquanto o ordenamento prossegue.

A maioria dos computadores pessoais usa uma interface gráfica GUI. Na realidade, a GUI é um programa sendo executado em cima do sistema operacional, como um shell. Nos sistemas Linux, o usuári escolhe pelo menos duas GUIs: Gnome e KDE ou nenhuma (usando uma janela de terminal no X11).

1.3 Chamadas de sistema

O protocolo padrão para chamadas de sistema é o POSIX (International Standard 9945-1). Nas seções a seguir, examinaremos algumas das chamadas de sistema POSIX mais usadas, ou mais especificamente, as rotinas de biblioteca que fazem uso dessas chamadas de sistema. POSIX tem cerca de 100 chamadas de rotina. Algumas das mais importantes estão listadas na Figura 1.18, agrupadas, por conveniência, em quatro categorias. Em grande medida, os serviços oferecidos por essas chamadas determinam a maior parte do que o sistema operacional tem de fazer, tendo em vista que o gerenciamento de recursos em computadores pessoais é mínimo (pelo menos comparado a grandes máquinas com múltiplos usuários). Os serviços incluem coisas como criar e finalizar processos, criar, excluir, ler e escrever arquivos, gerenciar diretórios e realizar entradas e saídas.

O mapeamento de chamadas de rotina POSIX em chamadas de sistema não é de uma para uma. O padrão POSIX especifica uma série de procedimentos que um sistema em conformidade com esse padrão deve oferecer, mas ele não especifica se elas são chamadas de sistema ou chamadas de biblioteca. Se uma rotina pode ser executada sem invocar uma chamada de sistema (isto é, sem um desvio para o núcleo), normalmente ela será realizada no espaço do usuário por questões de desempenho. No entanto, a maioria das rotinas POSIX invoca chamadas de sistema, em geral com uma rotina mapeando diretamente uma chamada de sistema.

FIGURA 1.18 Algumas das principais chamadas de sistema POSIX. O código de retorno *s* é –1 se um erro tiver ocorrido. Os códigos de retorno são os seguintes: *pid* é um processo id, *fd* é um descritor de arquivo, *n* é um contador de bytes, *position* é um deslocamento no interior do arquivo e *seconds* é o tempo decorrido. Os parâmetros são explicados no texto.

Gerenciamento de processos

Chamada	Descrição
pid = fork( )	Cria um processo filho idêntico ao pai
pid = waitpid(pid, &statloc, options)	Espera que um processo filho seja concluído
s = execve(name, argv, environp)	Substitui a imagem do núcleo de um processo
exit(status)	Conclui a execução do processo e devolve status

Gerenciamento de arquivos

Chamada	Descrição
fd = open(file, how, ...)	Abre um arquivo para leitura, escrita ou ambos
s = close(fd)	Fecha um arquivo aberto
n = read(fd, buffer, nbytes)	Lê dados a partir de um arquivo em um buffer
n = write(fd, buffer, nbytes)	Escreve dados a partir de um buffer em um arquivo
position = lseek(fd, offset, whence)	Move o ponteiro do arquivo
s = stat(name, &buf)	Obtém informações sobre um arquivo

Gerenciamento do sistema de diretório e arquivo

Chamada	Descrição
s = mkdir(name, mode)	Cria um novo diretório
s = rmdir(name)	Remove um diretório vazio
s = link(name1, name2)	Cria uma nova entrada, name2, apontando para name1
s = unlink(name)	Remove uma entrada de diretório
s = mount(special, name, flag)	Monta um sistema de arquivos
s = umount(special)	Desmonta um sistema de arquivos

Diversas

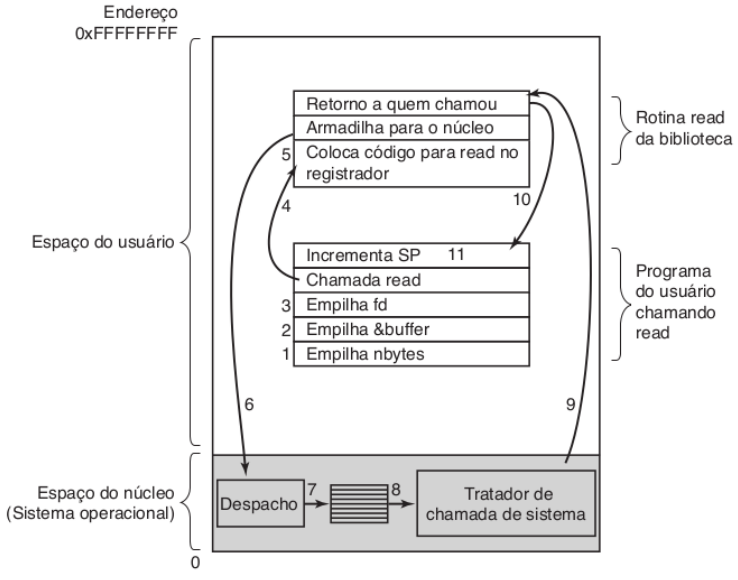
Chamada	Descrição
s = chdir(dirname)	Altera o diretório de trabalho
s = chmod(name, mode)	Altera os bits de proteção de um arquivo
s = kill(pid, signal)	Envia um sinal para um processo
seconds = time(&seconds)	Obtém o tempo decorrido desde 1º de janeiro de 1970

Computadores com uma única CPU podem executar apenas uma instrução de cada vez. Se um processo estiver executando um programa de usuário em modo de usuário e precisa de um serviço de sistema, como ler dados de um arquivo, ele tem de executar uma instrução de armadilha (*trap*) para transferir o controle para o sistema operacional (do modo usuário para o modo núcleo). O sistema operacional verifica os parâmetros e descobre o que o processo que está chamando quer. Então ele executa a chamada de sistema e retorna o controle para a instrução seguinte à chamada de sistema. De certa maneira, fazer uma chamada de sistema é como fazer um tipo especial de chamada de rotina, apenas que as chamadas de sistema entram no núcleo e as chamadas de rotina, não.

A chamada de sistema read, por exemplo, tem três parâmetros: o primeiro especificando o arquivo, o segundo é um ponteiro para o buffer e o terceiro dá o número de bytes a ser lido. Como

quase todas as chamadas de sistema, ele é invocado de programas C chamando uma rotina de biblioteca com o mesmo nome que a chamada de sistema: read. Uma chamada de um programa C pode parecer desta forma: contador = read(fd, buffer, nbytes). A chamada de sistema (e a rotina de biblioteca) retornam o número de bytes realmente lidos em contador. Esse valor é normalmente o mesmo que nbytes, mas pode ser menor, se, por exemplo, o caractere fim-de-arquivo (EOF) for encontrado durante a leitura. Se a chamada de sistema não puder ser realizada por causa de um parâmetro inválido ou de um erro de disco, o contador passa a valer −1, e o número de erro é colocado em uma variável global, errno. Os programas devem sempre conferir os resultados de uma chamada de sistema para ver se um erro ocorreu. Em preparação para chamar a rotina de biblioteca read, que na realidade é quem faz a chamada de sistema read , o programa de chamada primeiro empilha os parâmetros, como mostrado nos passos 1 a 3 na Figura 1.17. O primeiro e o terceiro parâmetros são chamados por valor, mas o segundo parâmetro é passado por referência, significando que o endereço do buffer (indicado por &) é passado, não seu conteúdo. Então vem a chamada real para a rotina de biblioteca (passo 4). A rotina de biblioteca, possivelmente escrita em linguagem de montagem, tipicamente coloca o número da chamada de sistema em um lugar onde o sistema operacional a espera, como um registro (passo 5). Então, ela executa uma instrução TRAP para passar do modo usuário para o modo núcleo e começar a execução em um endereço fixo dentro do núcleo (passo 6). A instrução TRAP não pode saltar para um endereço arbitrário. Dependendo da arquitetura, ela salta para um único local fixo, ou há um campo de 8 bits na instrução fornecendo o índice para uma tabela na memória contendo endereços para saltar. O código de núcleo que se inicia seguindo a instrução TRAP examina o número da chamada de sistema e então o despacha para o tratador correto da chamada de sistema, normalmente através de uma tabela de ponteiros que designam as rotinas de tratamento de chamadas de sistema indexadas pelo número da chamada (passo 7). Nesse ponto, é executado o tratamento de chamada de sistema (passo 8). Uma vez que ele tenha completado o seu trabalho, o controle pode ser retornado para a rotina de biblioteca no espaço do usuário na instrução após a instrução TRAP (passo 9). Essa rotina retorna para o programa do usuário da maneira usual que as chamadas de rotina retornam (passo 10). Para terminar a tarefa, o programa do usuário tem de limpar a pilha, como ele faz após qualquer chamada de rotina (passo 11). Presumindo que a pilha cresce para baixo, como muitas vezes é o caso, o código compilado incrementa o ponteiro da pilha exatamente o suficiente para remover os parâmetros empilhados antes da chamada read. O programa está livre agora para fazer o que quiser em seguida.

FIGURA 1.17 Os 11 passos na realização da chamada de sistema read (fd, buffer, nbytes).



1.3.1 Chamadas de sistema para gerenciamento de processos

A chamada fork é a única maneira para se criar um processo novo em POSIX. Ela cria uma cópia exata do processo original, incluindo todos os descritores de arquivos e registradores. Após a fork, o processo original e a cópia (o processo pai e o processo filho) seguem seus próprios caminhos separados. Todas as variáveis têm valores idênticos no momento da fork, mas como os dados do processo pai são copiados para criar o processo filho, mudanças subsequentes em um deles não afetam o outro. Já, o texto do programa (código) é inalterável, tanto no processo pai, quanto no processo filho. A chamada fork retorna um valor, que é zero no processo filho e igual ao PID (Process IDentifier — identificador de processo) do processo filho no processo pai. Usando o PID retornado, os dois processos podem ver qual é o processo pai e qual é o filho.

Quando um comando é digitado, o shell cria um novo processo. Esse processo filho tem de executar o comando de usuário. Ele o faz usando a chamada de sistema execve. Um shell altamente simplificado ilustrando o uso de fork, waitpid e execve é mostrado na Figura 1.19. Execve possui três parâmetros: o nome do arquivo a ser executado, um ponteiro para o arranjo de argumentos e um ponteiro para o arranjo de ambiente. Vamos considerar o caso de um comando como:

cp fd1 fd2

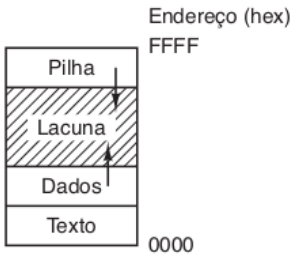
usado para copiar o fd1 para o fd2. Após o shell ter criado o processo filho, este localiza e executa o arquivo cp e passa para ele os nomes dos arquivos de origem e de destino. O programa principal de cp contém a declaração main(argc, argv, envp) onde argc é uma contagem do número de itens na linha de comando, incluindo o nome do programa. O segundo parâmetro, argv, é um ponteiro para um arranjo. Em nosso exemplo, argv[0] apontaria para a cadeia de caracteres “cp”, argv[1] apontaria para a “fd1” e argv[2] apontaria para a “fd2”. O terceiro parâmetro do main, envp, é um ponteiro para o ambiente, um arranjo de cadeias de caracteres contendo atribuições da forma nome = valor usadas para passar informações como o tipo de terminal e o nome do diretório home para programas. Se nenhum ambiente é passado para o processo filho, então o terceiro parâmetro de execve é um zero.

FIGURA 1.19 Um interpretador de comandos simplificado. Neste livro, presume-se que *TRUE* seja definido como 1.*

```
#define TRUE 1

while (TRUE) {                                  /* repita para sempre */
      type_prompt( );                           /* mostra prompt na tela */
      read_command(command, parameters);        /* le entrada do terminal */

      if (fork( ) != 0) {                       /* cria processo filho */
          /* Codigo do processo pai. */
          waitpid(–1, &status, 0);              /* aguarda o processo filho acabar */
      } else {
          /* Codigo do processo filho. */
          execve(command, parameters, 0);       /* executa o comando */
      }
}
```

Processos em UNIX têm sua memória dividida em três segmentos: o segmento de texto (isto é, código de programa), o segmento de dados (isto é, as variáveis) e o segmento de pilha. O segmento de dados cresce para cima e a pilha cresce para baixo, como mostrado na Figura 1.20. Entre eles há uma lacuna de espaço de endereço não utilizado. A pilha cresce na lacuna automaticamente, na medida do necessário.

FIGURA 1.20 Os processos têm três segmentos: texto, dados e pilha.



1.3.2 Chamadas de sistema para gerenciamento de arquivos

Para ler ou escrever um arquivo, é preciso primeiro abri-lo. Essa chamada especifica o nome do arquivo a ser aberto, seja como um nome de caminho absoluto ou relativo ao diretório de trabalho, assim como um código de O_RDONLY, O_WRONLY, ou O_RDWR, significando aberto para leitura, escrita ou ambos. Para criar um novo arquivo, o parâmetro O_CREAT é usado. O descritor de arquivos retornado pode então ser usado para leitura ou escrita. Em seguida, o arquivo pode ser fechado por close, que torna o descritor disponível para ser reutilizado em um open subsequente. As chamadas mais intensamente usadas são read e write.

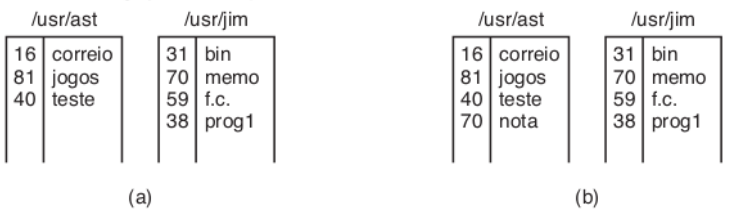
Associado a cada arquivo há um ponteiro que indica a posição atual no arquivo. Quando lendo (escrevendo) sequencialmente, ele em geral aponta para o próximo byte a ser lido (escrito). A chamada lseek muda o valor do ponteiro de posição, de maneira que chamadas subsequentes para ler ou escrever podem começar em qualquer parte no arquivo. Lseek tem três parâmetros: o primeiro é o descritor de arquivo, o segundo é uma posição do arquivo e o terceiro diz se a posição do arquivo é relativa ao começo, à posição atual ou ao fim do arquivo. O valor retornado por lseek é a posição absoluta no arquivo (em bytes) após mudar o ponteiro.

1.3.3 Chamadas de sistema para gerenciamento de diretórios

As primeiras duas chamadas, mkdir e rmdir, criam e removem diretórios vazios, respectivamente. A próxima chamada é link. Sua finalidade é permitir que o mesmo arquivo apareça sob dois ou mais nomes, muitas vezes em diretórios diferentes. Um uso típico é permitir que vários membros da mesma equipe de programação compartilhem um arquivo comum, com cada um deles tendo o arquivo aparecendo no seu próprio diretório, possivelmente sob nomes diferentes. Compartilhar um arquivo não é o mesmo que dar a cada membro da equipe uma cópia particular; ter um arquivo compartilhado significa que as mudanças feitas por qualquer membro da equipe são instantaneamente visíveis para os outros membros, mas há apenas um arquivo.

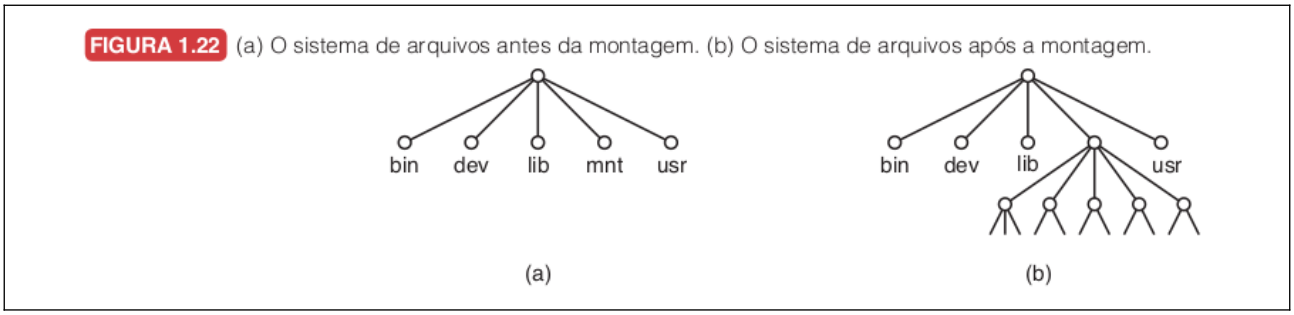
link(“/usr/jim/memo”, “/usr/ast/note”);

FIGURA 1.21 (a) Dois diretórios antes da ligação de */usr/jim/memo* ao diretório *ast*. (b) Os mesmos diretórios depois dessa ligação.



A chamada de sistema mount permite que dois sistemas de arquivos sejam fundidos em um. Ao executar a chamada de sistema mount, o sistema de arquivos USB pode ser anexado ao sistema

de arquivos-raiz, como mostrado na Figura 1.22. Um comando típico em C para realizar essa montagem é mount(“/dev/sdb0”, “/mnt”, 0); onde o primeiro parâmetro é o nome de um arquivo especial de blocos para a unidade de disco 0, o segundo é o lugar na árvore onde ele deve ser montado, e o terceiro diz se o sistema de arquivos deve ser montado como leitura e escrita ou somente leitura. A chamada mount torna possível integrar mídias removíveis (pendrives, por exemplo) em uma única hierarquia de arquivos integrada, sem precisar preocupar-se em qual dispositivo se encontra um arquivo. Quando um sistema de arquivos não é mais necessário, ele pode ser desmontado com a chamada de sistema umount.



1.3.4 Chamadas de sistema diversas

A chamada chdir (cd no Linux) muda o diretório de trabalho atual. Após a chamada chdir(“/usr/ast/test”); uma abertura no arquivo xyz abrirá /usr/ast/test/xyz. A chamada de sistema chmod torna possível mudar o modo de um arquivo. Por exemplo, para tornar um arquivo como somente de leitura para todos, exceto o proprietário, poderia ser executado chmod(“file”, 0644). Com a chamada de sistema kill, um sinal é enviado para matar um processo. O POSIX define uma série de rotinas para lidar com o tempo. Por exemplo, time retorna o tempo atual somente em segundos, com 0 correspondendo a 1º de janeiro, 1970, à meia-noite.

1.3.5 A API Win32 do Windows

O Windows e o UNIX diferem de uma maneira fundamental em seus respectivos modelos de programação. O Windows também tem chamadas de sistema. Com o UNIX, há quase uma relação de um para um entre as chamadas de sistema (por exemplo, read) e as rotinas de biblioteca (por exemplo, read) usadas para invocar as chamadas de sistema. Em outras palavras, para cada chamada de sistema, há aproximadamente uma rotina de biblioteca que é chamada para invocá-la.

Com o Windows, a situação é radicalmente diferente. As chamadas de biblioteca e as chamadas de sistema reais são altamente desacopladas. A Microsoft definiu um conjunto de rotinas chamadas de API Win32 (Application Programming Interface — interface de programação de aplicativos) que se espera que os programadores usem para acessar os serviços do sistema operacional. Essa interface tem contado com o suporte (parcial) de todas as versões do Windows desde o Windows 95. A Win32 proporciona compatibilidade entre as versões do Windows.

O número de chamadas API Win32 é extremamente grande, chegando a milhares. Além disso, enquanto muitas delas invocam chamadas de sistema, um número substancial é executado inteiramente no espaço do usuário. Como consequência, com o Windows é impossível de se ver o que é uma chamada de sistema (isto é, realizada pelo núcleo) e o que é apenas uma chamada de biblioteca do espaço do usuário. Ou seja, nem todas elas são verdadeiras chamadas de sistema (isto é, levam o controle para o núcleo).

1.4 Estrutura de sistemas operacionais

Neste tópico serão analisados seis projetos de estrutura de sistemas operacionais que foram implementados na prática para transmitir uma ideia do espectro de possibilidades.

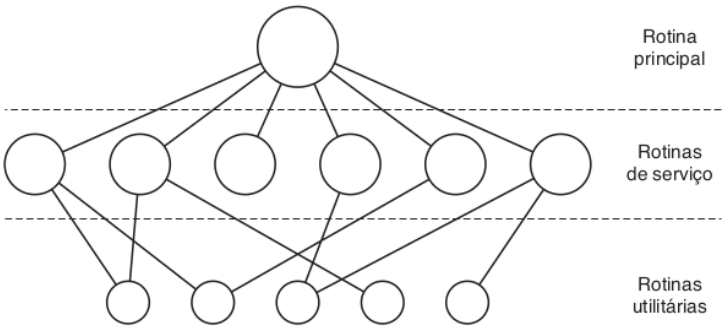
1.4.1 Sistemas monolíticos

Organização mais comum. Todo sistema operacional é executado como um único programa em modo núcleo. É escrito como uma coleção de rotinas, ligadas a um único grande programa binário executável. Cada procedimento no sistema é livre para chamar qualquer outro. Isso pode levar a um sistema difícil de lidar e compreender. Além disso, uma quebra em qualquer uma dessas rotinas derrubará todo o sistema operacional.

Para preparar o SO, é preciso primeiro compilar todas os arquivos contendo as rotinas, juntando-as num único arquivo executável usando o ligador (linker) do sistema. As rotinas são muito acopladas (oposta a uma estrutura modularizada e coesa). Apesar disso, o projeto de sistema operacional de estrutura monolítica segue uma lógica. As chamadas de sistema são requisitadas, utilizando parâmetros de entrada empilhados num local bem definido. Posteriormente, executa-se a instrução de desvio de controle (trap) que chaveia a máquina do modo usuário para o modo núcleo e transfere o controle para o sistema operacional. O SO, por sua vez, busca os parâmetros e determina qual chamada de sistema será executada. Depois disso, ele indexa uma tabela que contém na linha k um ponteiro para a rotina que executa a chamada de sistema k (Passo 7: Despacho da Figura 1.17).

Assim, a estrutura em questão é organizada da seguinte forma: (1) Programa principal: invoca as rotinas de serviço requisitadas; (2) Rotinas de serviço: executam as chamadas de sistema; (3) Rotinas utilitárias: auxiliam as rotinas de serviço (ex.: buscam dados de programas dos usuários).

FIGURA 1.24 Um modelo de estruturação simples para um sistema monolítico.



1.4.2 Sistemas de camadas

A organização do sistema operacional como uma hierarquia de camadas foi implementado pela primeira vez em 1968. O chamado “Sistema THE”, desenvolvido por Dijkstra e seus alunos, tinha 6 camadas, conforme descritas na figura seguinte.

FIGURA 1.25 Estrutura do sistema operacional THE.

Camada	Função
5	O operador
4	Programas de usuário
3	Gerenciamento de entrada/saída
2	Comunicação operador–processo
1	Memória e gerenciamento de tambor
0	Alocação do processador e multiprogramação

A camada 0 lidava com a alocação do processador, realizando o chaveamento de processos quando ocorriam interrupções ou quando os temporizadores expiravam. Acima da camada 0, o sistema consistia em processos sequenciais e cada um deles podia ser programado sem precisar preocupar-se com o fato de que múltiplos processos estavam sendo executados em um único processador. Em outras palavras, a camada 0 fornecia a multiprogramação básica da CPU.

A camada 1 realizava o gerenciamento de memória. Ela alocava espaço para processos na memória principal e em um tambor magnético usado para armazenar partes de processos (páginas). O software da camada 1 certificava-se de que as páginas fossem trazidas à memória no momento em que eram necessárias e removidas quando não eram mais. A camada 2 encarregava-se da comunicação entre cada processo. A camada 3 encarregava-se do gerenciamento dos dispositivos de E/S e armazenava temporariamente os fluxos de informação que iam ou vinham desses dispositivos. Acima da camada 3, cada processo podia lidar com dispositivos de E/S abstratos mais acessíveis, em vez de dispositivos reais com muitas peculiaridades. A camada 4, os programas dos usuários não precisavam se preocupar com o gerenciamento de processo, memória, console ou E/S.

1.4.3 Micronúcleos

Qual deve ser o limite entre o usuário e o núcleo? A abordagem de camadas levantou o questionamento de onde seria adequado traçar este limite. O fato é que erros no código do núcleo podem derrubar o sistema instantaneamente. Isso justificaria a colocação do mínimo de funcionalidades possíveis no modo núcleo. Assim, processos de usuário poderiam ser configurados para ter menos poder, de maneira que um erro possa não ser fatal.

Sistemas operacionais são a tal ponto sujeitos a erros, que os fabricantes de computadores colocam botões de reinicialização neles (muitas vezes no painel da frente), algo que os fabricantes de TVs, aparelhos de som e carros não o fazem, apesar da grande quantidade de software nesses dispositivos.

A ideia básica por trás do projeto de micronúcleo é atingir uma alta confiabilidade através da divisão do sistema operacional em módulos pequenos e bem definidos, apenas um dos quais — o micronúcleo — é executado em modo núcleo e o resto é executado como processos de usuário comuns relativamente sem poder.

Em particular, ao se executar cada driver de dispositivo e sistema de arquivos como um processo de usuário em separado, um erro em um deles pode derrubar esse componente, mas não consegue derrubar o sistema inteiro. Desse modo, um erro no driver de áudio fará que o som fique truncado ou pare, mas não derrubará o computador. Em comparação, em um sistema monolítico, com todos os drivers no núcleo, um driver de áudio com problemas pode facilmente referenciar um endereço de memória inválido e provocar uma parada instantânea no sistema.

Com a exceção do OS X, que é baseado no micronúcleo Mach, sistemas operacionais de computadores de mesa comuns não usam micronúcleos. No entanto, eles são dominantes em aplicações de tempo real, industriais, de aviônica e militares, que são cruciais para missões e têm exigências de confiabilidade muito altas.

O micronúcleo MINIX 3, por exemplo, levou a ideia da modularidade até o limite, decompondo a maior parte do sistema operacional em uma série de processos de modo usuário independentes. MINIX 3 é um sistema em conformidade com o POSIX, de código aberto e gratuitamente disponível em <www.minix3.org>. O MINIX 3 tem muitas restrições limitando o poder de cada processo. Como mencionado, os drivers podem tocar apenas portas de E/S autorizadas, mas o acesso às chamadas de núcleo também é controlado processo a processo, assim como a capacidade de enviar mensagens para outros processos. Processos também podem conceder uma permissão limitada para outros processos para que o núcleo acesse seus espaços de endereçamento. A soma de todas essas restrições é que cada driver e servidor têm exatamente o poder de fazer o seu trabalho e nada mais, dessa maneira limitando muito o dano que um componente com erro pode provocar.

Uma ideia de certa maneira relacionada a ter um núcleo mínimo é colocar o mecanismo para fazer algo no núcleo, mas não a política. Para esclarecer esse ponto, considere o escalonamento de processos. Um algoritmo de escalonamento relativamente simples é designar uma prioridade numérica para todo processo e então fazer com que o núcleo execute o processo mais prioritário e que seja executável. O mecanismo — no núcleo — é procurar pelo processo mais prioritário e executá-lo. A política — designar prioridades para processos — pode ser implementada por processos de modo usuário. Dessa maneira, política e mecanismo podem ser desacoplados. Assim, o núcleo torna-se menor.

1.4.4 O modelo cliente-servidor

Uma ligeira variação da ideia do micronúcleo é distinguir duas classes de processos, os servidores, que prestam algum serviço, e os clientes, que usam esses serviços. Esse modelo é conhecido como o modelo cliente-servidor. A comunicação entre clientes e servidores é realizada muitas vezes pela troca de mensagens. Para obter um serviço, um processo cliente constrói uma mensagem dizendo o que ele quer e a envia ao serviço apropriado. O serviço então realiza o trabalho e envia de volta a resposta. Desse modo, o modelo cliente-servidor é uma abstração que pode ser usada para uma única máquina ou para uma rede de máquinas.

1.4.5 Máquinas virtuais

Empresas tradicionais executavam seus próprios servidores de correio, de web, de FTP e outros servidores em computadores separados, às vezes com sistemas operacionais diferentes. Elas veem a virtualização como uma maneira de executar todos eles por meio da virtualização.

Sem a virtualização, os clientes de hospedagem na web são obrigados a escolher entre a hospedagem compartilhada (que dá a eles uma conta de acesso a um servidor da web, mas nenhum controle sobre o software do servidor) e a hospedagem dedicada (que dá a eles a própria máquina, que é muito flexível, mas cara para sites de pequeno a médio porte).

Quando uma empresa de hospedagem na web oferece máquinas virtuais para alugar, uma única máquina física pode executar muitas máquinas virtuais, e cada uma delas parece ser uma máquina completa. Clientes que alugam uma máquina virtual podem executar qualquer sistema operacional e software que eles quiserem, mas a uma fração do custo de um servidor dedicado (pois a mesma máquina física dá suporte a muitas máquinas virtuais ao mesmo tempo).

Outro uso da virtualização é por usuários finais que querem poder executar dois ou mais sistemas operacionais ao mesmo tempo, digamos Windows e Linux, pois alguns dos seus pacotes de aplicativos favoritos são executados em um sistema e outros no outro sistema.

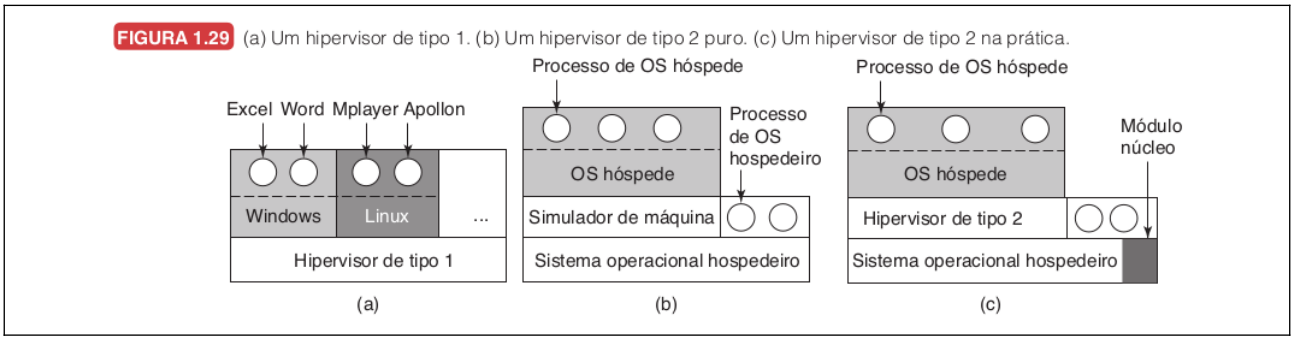
A fim de executar um software de máquina virtual em um computador, a sua CPU tem de ser virtualizável. Hipervisores populares hoje em dia incluem KVM (para o núcleo Linux), VirtualBox (da Oracle) e Hyper-V (da Microsoft).

Na prática, a distinção real entre um hipervisor tipo 1 e um hipervisor tipo 2 é que o tipo 2 usa um sistema operacional hospedeiro e o seu sistema de arquivos para criar processos, armazenar arquivos e assim por diante. Um hipervisor tipo 1 não tem suporte subjacente e precisa realizar todas essas funções sozinho.

Após um hipervisor tipo 2 ser inicializado, ele lê o CD-ROM de instalação (ou arquivo de imagem CD-ROM) para o sistema operacional hóspede escolhido e o instala em um disco virtual, que é apenas um grande arquivo no sistema de arquivos do sistema operacional hospedeiro. Hipervisores tipo 1 não podem realizar isso porque não há um sistema operacional hospedeiro para armazenar os arquivos. Eles têm de gerenciar sua própria armazenagem em uma partição de disco bruta.

Quando o sistema operacional hóspede é inicializado, ele faz o mesmo que no hardware de verdade, tipicamente iniciando alguns processos de segundo plano e então uma interface gráfica

GUI. Para o usuário, o sistema operacional hóspede comporta-se como quando está sendo executado diretamente no hardware, embora não seja o caso aqui.



1.4.6 Exonúcleos

Em vez de clonar a máquina real, como é feito com as máquinas virtuais, outra estratégia é dividi-la, concedendo a cada usuário um subconjunto dos recursos. Desse modo, uma máquina virtual pode obter os blocos de disco de 0 a 1.023, a próxima pode ficar com os blocos 1.024 a 2.047 e assim por diante. Na camada de baixo, executando em modo núcleo, há um programa chamado exonúcleo. Sua tarefa é alocar recursos às máquinas virtuais e então conferir tentativas de usá-las para assegurar-se de que nenhuma máquina esteja tentando usar os recursos de outra. Cada máquina virtual no nível do usuário pode executar seu próprio sistema operacional. No entanto, cada uma está restrita a usar apenas os recursos que ela pediu e foram alocados.

A vantagem do esquema do exonúcleo é que ele poupa uma camada de mapeamento. Nos outros projetos, cada máquina virtual pensa que ela tem seu próprio disco, com blocos sendo executados de 0 a algum máximo, de maneira que o monitor da máquina virtual tem de manter tabelas para remapear os endereços de discos (e todos os outros recursos). Com o exonúcleo, esse remapeamento não é necessário. O exonúcleo precisa apenas manter o registro de para qual máquina virtual foi atribuído qual recurso. Esse método ainda tem a vantagem de separar a multiprogramação (no exonúcleo) do código do sistema operacional do usuário (em espaço do usuário), mas com menos sobrecarga, tendo em vista que tudo o que o exonúcleo precisa fazer é manter as máquinas virtuais distantes umas das outras.

Cap. 2: Processos e Threads

Processo é o conceito central do Sistema Operacional. Tudo depende dele. Trata-se de uma abstração que o SO proporciona. Eles suportam operações (pseudo) concorrentes mesmo quando há apenas uma única CPU disponível, transformando-a em múltiplas CPUs virtuais. Sem a abstração de processo, a computação moderna não poderia existir.

2.1 Processos

Computadores realizam tarefas simultâneas. Resposta de requisições aos servidores web chegam de toda parte. Quando uma requisição chega, o servidor confere para ver se a página requisitada está em *cache*. Se estiver, ela responde ao cliente; se não, uma solicitação de acesso ao disco é iniciada para buscá-la. No entanto, tal solicitação é lenta, comparada com a velocidade de processamento da CPU. Enquanto a solicitação de acesso ao disco é aguardada, outras podem chegar. Por isso, algum método é necessário para modelar e controlar essa concorrência. Processos (especialmente *threads*) ajudam nisso.

Quando o sistema é inicializado, muitos processos são iniciados: uns de forma implícita (ex.: esperar pela chegada de e-mails; antivírus conferindo se há definição de novos vírus, etc) e outros de forma explícita (ex.: imprimir arquivo; salvar foto num pendrive, etc). Tudo isso pode ocorrer enquanto o usuário acessa a internet, usando um navegador web, por exemplo. Toda essa atividade é gerenciada com um sistema de multiprogramação dando suporte a múltiplos processos.

Em sistema de multiprogramação, a CPU muda de um processo para outro rapidamente, executando cada um por dezenas ou centenas de milissegundos. Enquanto, em qualquer dado instante a CPU está executando apenas um processo, no curso de 1s ela pode trabalhar em vários deles, dando a ilusão do paralelismo. O pseudoparalelismo diferencia-se do verdadeiro paralelismo de hardware nos sistemas multiprocessadores (que têm duas ou mais CPUs compartilhando a mesma memória física).

2.1.1 O modelo de processo

Todos os softwares executáveis no computador, são organizados em uma série de processos sequenciais, ou, simplesmente, processos. Processo é uma instância de programa em execução, incluindo valores atuais do contador do programa, registradores e variáveis. A CPU troca a todo momento de processo em processo. Os processos são executados de forma pseudoparalela. Esse mecanismo de trocas rápidas é chamado de multiprogramação. Na Fig. 2.1(b) há 4 processos, cada um com seu próprio fluxo de controle (isto é, seu próprio contador de programa lógico), sendo executado independente dos outros. Há apenas um contador de programa físico. Assim, quando cada processo é executado, o seu contador de programa lógico é carregado para o contador de programa real. No momento em que ele é concluído, o contador de programa físico é salvo no contador de programa lógico do processo na memória. Na Fig. 2.1(c) vemos que, analisados durante um intervalo longo o suficiente, todos os processos tiveram progresso, mas a qualquer dado instante apenas um está sendo de fato executado.

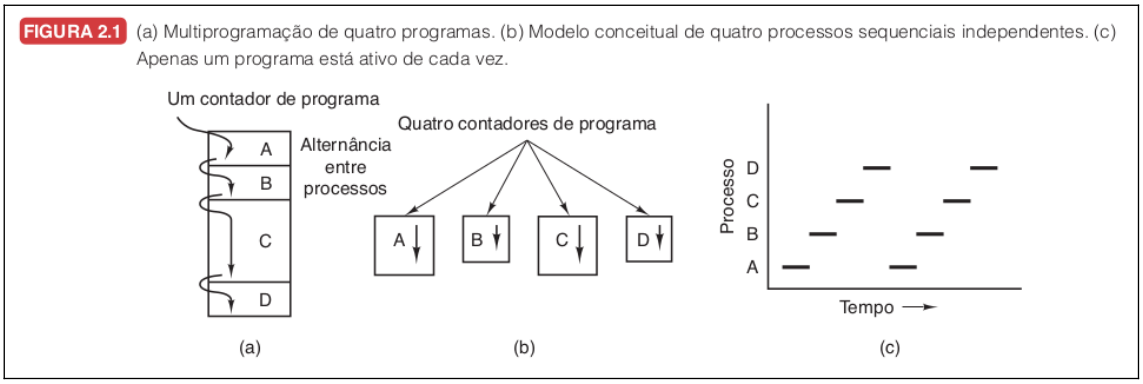


FIGURA 2.1 (a) Multiprogramação de quatro programas. (b) Modelo conceitual de quatro processos sequenciais independentes. (c) Apenas um programa está ativo de cada vez.

Neste capítulo, presumiremos que há apenas uma única CPU. No entanto, essa suposição não é verdadeira atualmente, tendo em vista que os chips novos são muitas vezes multinúcleos (*multicore*), com dois ou mais núcleos. Uma única CPU pode executar apenas um processo de cada vez, se há dois núcleos (ou CPUs), em cada um deles pode ser executado apenas um processo de cada vez.

Com o chaveamento rápido da CPU entre os processos, a taxa pela qual um processo realiza a sua computação não será uniforme e provavelmente nem reproduzível se os mesmos processos forem executados outra vez. Desse modo, processos não devem ser programados com suposições predefinidas sobre a temporização. Ex.: Processo de áudio e processo de vídeo. Neste caso, o

temporizador precisa ser confiável para que haja sincronia ente aúdio e vídeo. Quando um processo tem exigências de tempo real, críticas como essa, eventos particulares, têm de ocorrer dentro de um número específico de milissegundos e medidas especiais precisam ser tomadas para assegurar que elas ocorram. Em geral, no entanto, a maioria dos processos não é afetada pela multiprogramação subjacente da CPU ou as velocidades relativas de processos diferentes.

A diferença entre um processo e um programa é sutil, mas absolutamente crucial. Um processo é uma atividade de algum tipo. Ela tem um programa, uma entrada, uma saída e um estado. Um único processador pode ser compartilhado entre vários processos, com algum algoritmo de escalonamento sendo usado para determinar quando parar o trabalho em um processo e servir outro. Em comparação, um programa é algo que pode ser armazenado em disco sem fazer nada.

Se um programa está sendo executado duas vezes, é contado como dois processos. O fato de que dois processos em execução estão operando o mesmo programa não importa, eles são processos distintos. O sistema operacional pode ser capaz de compartilhar o código entre eles de maneira que apenas uma cópia esteja na memória.

2.1.2 Criação

Em sistemas projetados para executar apenas uma única aplicação (ex.: controlador de forno micro-ondas), é possível ter todos os processos que serão necessários, tão logo o sistema seja ligado. Por outro lado, em sistemas para fins gerais, é necessário criar e terminar processos, na medida do necessário, durante a operação. Exemplos de eventos que criam processos:

1. **Inicialização do sistema:** quando um sistema operacional é inicializado, uma série de processos é criada. Alguns deles são de primeiro plano, isto é, processos que interagem com usuários (humanos) e realizam trabalho para eles. Outros operam no segundo plano e não estão associados com usuários, mas têm alguma função específica. Processos que ficam em segundo plano para lidar com atividades como e-mail, páginas da web, notícias, impressão, etc., são chamados de *daemons*.

2. **Execução de uma chamada de sistema de criação de processo por um processo em execução:** além dos processos de inicialização do sistema, novos processos podem ser criados, emitindo chamadas de sistema para criar um ou mais processos novos. Criar processos novos é útil quando o trabalho pode ser formulado como vários processos relacionados, interagindo de maneira independente. Num multiprocessador, permitir que cada processo execute em uma CPU diferente também pode fazer com que a tarefa seja realizada mais rápido.

3. **Solicitação de um usuário para criar um novo processo:** usuários podem começar um programa digitando um comando ou clicando duas vezes sobre um ícone. Cada uma dessas ações inicia um novo processo e executa nele o programa selecionado. Ou seja, o usuário pode interagir com o processo, fornecendo a entrada quando necessário.

4. **Início de uma tarefa em lote:** no gerenciamento de estoque ao fim de um dia numa rede de lojas, por exemplo, podem ser submetidas tarefas em lote ao sistema. Ou seja, quando o sistema operacional decide que ele tem os recursos para executar outra tarefa, ele cria um novo processo e executa a próxima tarefa a partir da fila de entrada.

Em todos esses casos, um novo processo é criado por outro já existente executando uma chamada de sistema de criação de processo. Essa chamada de sistema diz ao sistema operacional para criar um novo processo e indica, direta ou indiretamente, qual programa executar nele.

No UNIX, há apenas uma chamada de sistema para criar um novo processo: fork. Ela cria um clone do processo que a chamou. Assim, ambos os processos, o pai e o filho, têm a mesmas imagens de memória, variáveis de ambiente e os mesmos arquivos abertos. O processo filho executa execve ou uma chamada de sistema similar para executar uma nova instância do programa. O objetivo desse processo é permitir que o processo filho manipule seus descritores de arquivos depois da fork, mas antes da execve, a fim de conseguir o redirecionamento de entrada padrão, saída padrão e erro padrão.

No Windows, além do CreateProcess (que pode receber até 10 parâmetros de entrada), o Win32 tem aproximadamente 100 outras funções para gerenciar e sincronizar processos e tópicos relacionados.

2.1.3 Término

Após um processo ter sido criado, ele começa a ser executado e realiza seu trabalho. Seu término pode ocorrer devido a uma dessas condições:

1. **Saída normal (voluntária):** a maioria dos processos termina por terem realizado seu trabalho. Essa chamada é *exit* em UNIX e *ExitProcess* no Windows.
2. **Erro fatal (involuntário):** se um usuário digita o comando para compilar um programa e o arquivo não existe, o compilador anuncia esse fato e termina a execução.
3. **Saída por erro (voluntária):** causado pelo processo, devido a um erro de programa. Ex.: executar instrução ilegal; referenciar memória inexistente; dividir por zero, etc.
4. **Morto por outro processo (involuntário):** o processo executa uma chamada de sistema dizendo ao sistema operacional para matar outro processo. Em UNIX, essa chamada é *kill*. A função Win32 correspondente é *TerminateProcess*. Em alguns sistemas, quando um processo é finalizado, todos os processos que ele criou são mortos.

2.1.4 Hierarquias

Em UNIX, quando um processo cria outro, o processo pai e o processo filho continuam a ser associados. O processo filho também pode criar mais processos, formando uma hierarquia. Cada processo pode pegar um sinal, ignorá-lo, ou assumir uma ação predefinida. Ao ligar um computador com um sistema operacional UNIX, por exemplo, um processo especial, chamado *init*, está presente na imagem de inicialização. Quando começa a ser executado, ele lê um arquivo dizendo quantos terminais existem, então ele se bifurca em um novo processo para cada terminal. Esses processos esperam que alguém se conecte. Se uma conexão é bem-sucedida, o processo de conexão executa um *shell* para aceitar comandos. Esses comandos podem iniciar mais processos e assim por diante. Desse modo, todos os processos no sistema inteiro pertencem a uma única árvore, com *init* em sua raiz.

O Windows não tem conceito de uma hierarquia de processos. Todos os processos são iguais. O único indício de uma hierarquia ocorre quando um processo é criado e o pai recebe um identificador especial (chamado de *handle*) que ele pode usar para controlar o filho. No entanto, ele

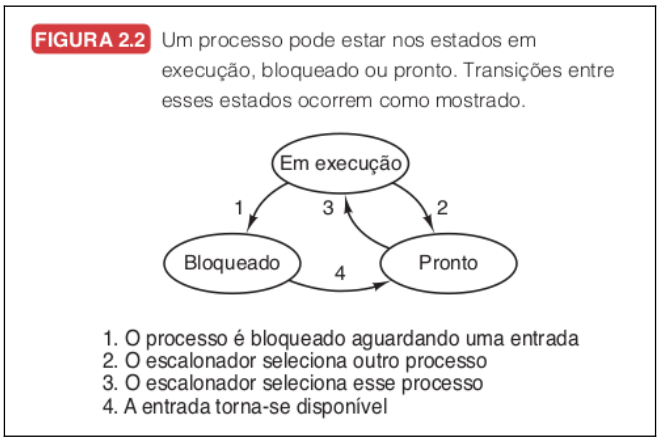
é livre para passar esse identificador para algum outro processo, invalidando a hierarquia. Processos em UNIX não podem deserdar seus filhos.

2.1.5 Estados

Cada processo é uma entidade independente, com seu próprio contador de programa e estado interno. No entanto, processos precisam interagir. Um processo pode gerar alguma saída que outro usa como entrada. Dependendo das velocidades relativas dos dois processos (que dependem tanto da complexidade relativa dos programas, quanto do tempo de CPU que cada um teve), pode acontecer do processo subsequente estar preparado para ser executado, mas não haver entrada esperando por ele. Ele deve então ser bloqueado até que alguma entrada esteja disponível. É possível, também, que um processo esteja pronto e capaz de executar. No entanto, ele pode ser bloqueado porque o sistema operacional decidiu alocar a CPU para outro processo por um tempo.

Há 3 estados nos quais um processo pode se encontrar:

1. **Em execução:** usando a CPU naquele instante;
2. **Pronto:** executável, porém, temporariamente parado para deixar outro processo ser executado. Ou seja, não há uma CPU disponível para ele;
3. **Bloqueado:** incapaz de ser executado, até que haja um entrada disponível. Ou seja, mesmo que a CPU esteja ociosa, ele permanece neste estado.



Há 4 transições entre estes estados:

1. Uma chamada de sistema bloqueia o processo em execução quando não há uma entrada disponível;
2. O escalonador de processos decide que o processo em andamento foi executado por tempo suficiente, e é o momento de deixar outro processo ter algum tempo de CPU;
3. O escalonador de processos decide que todos os outros processos tiveram sua parcela justa de utilização da CPU e desta vez, está na hora do primeiro processo voltar a ser executado pela CPU.
4. Ocorre quando o evento externo pelo qual um processo estava esperando (como a chegada de alguma entrada) acontece.

O nível mais baixo do sistema operacional é o escalonador, com uma variedade de processos acima dele. Todo o tratamento de interrupções e detalhes sobre o início e parada de processos estão ocultos no escalonador.

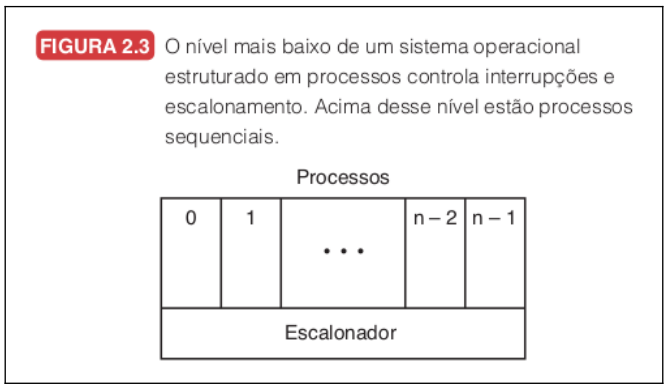


FIGURA 2.3 O nível mais baixo de um sistema operacional estruturado em processos controla interrupções e escalonamento. Acima desse nível estão processos sequenciais.

2.1.6 Implementação

Para implementar o modelo de processos, o sistema operacional mantém uma tabela de processos (ou blocos de controle de processo), com uma entrada para cada um deles. Essas entradas contêm informações sobre o estado do processo, incluindo o seu contador de programa, ponteiro de pilha, alocação de memória, estado dos arquivos abertos, informação sobre sua contabilidade e escalonamento e sobre trocas do estado em execução para pronto ou bloqueado, de maneira que ele possa ser reiniciado. A Figura 2.4 mostra alguns dos campos fundamentais em um sistema típico.

FIGURA 2.4 Alguns dos campos de uma entrada típica na tabela de processos.

Gerenciamento de processo	Gerenciamento de memória	Gerenciamento de arquivo
Registros	Ponteiro para informações sobre o segmento de texto	Diretório-raiz
Contador de programa		Diretório de trabalho
Palavra de estado do programa	Ponteiro para informações sobre o segmento de dados	Descritores de arquivo
Ponteiro da pilha		ID do usuário
Estado do processo	Ponteiro para informações sobre o segmento de pilha	ID do grupo
Prioridade		
Parâmetros de escalonamento		
ID do processo		
Processo pai		
Grupo de processo		
Sinais		
Momento em que um processo foi iniciado		
Tempo de CPU usado		
Tempo de CPU do processo filho		
Tempo do alarme seguinte		

Detalhes envolvendo a implementação de processos:

https://www.geeksforgeeks.org/process-table-and-process-control-block-pcb/

2.1.7 Modelando a multiprogramação

Há processos que utilizam muito recurso de CPU. São os chamados processos CPU-bound (orientados à CPU). Neste caso, o tempo de execução é definido pelos ciclos de processador. Em outros processos, utiliza-se muito do recurso de chamadas de sistema de Entrada-Saída. Estes últimos, são os processos I/O-bound (orientados à E/S). Neste caso, o tempo de execução é definido pela duração das operações de entrada e saída. O modelo ideal de multiprogramação é que exista um balanceamento entre processos CPU-bound e I/O-bound.

2.2 Threads

Cada processo tem um espaço de endereçamento e uma única *thread* (linha de execução) de controle. Em muitas situações, é desejável ter múltiplas *threads* de controle no mesmo espaço de endereçamento executando em quase paralelo, como se eles fossem (quase) processos separados (exceto pelo espaço de endereçamento compartilhado).

2.2.1 Utilização de threads

Qual é a finalidade de haver um tipo de processo dentro de um processo? Há razões para a existência desses miniprocessos, chamados *threads*. Num cenário onde múltiplas atividades ocorrem de forma simultânea, algumas delas poderiam, eventualmente, ser bloqueadas. Ao decompor uma aplicação dessas em múltiplas *threads* sequenciais que são executadas em quase paralelo, o modelo de programação torna-se mais simples. Isso viabiliza a capacidade para entidades em paralelo compartilharem um único espaço de endereçamento e todos os seus dados entre si. Ou seja, em vez de pensar a respeito de interrupções, temporizadores e chaveamentos de contextos, pode-se pensar a respeito de linhas de execução em paralelo. Como thread é mais leve do que processo, é mais fácil e rápido criá-la e destruí-la. Em muitos sistemas, criar uma *thread* é de 10 a 100 vezes mais rápido do que criar um processo. Aplicações envolvendo atividades substanciais de entrada e saída podem ser aceleradas com o uso de *threads*. Elas também são úteis em sistemas com múltiplas CPUs, onde o paralelismo real é possível.

Num editor de texto, por exemplo, o usuário apaga uma frase da página 1 de um livro de 800 páginas. De forma adicional, pode haver uma atividade sendo executada em lote, para substituir um termo indesejado. Posteriormente, ele digita um comando para que o editor de texto vá até a página 600. O editor é forçado a reformatar o livro inteiro até a página 600, algo difícil. Pode haver um atraso substancial antes que a página desejada seja exibida.

Threads são úteis nesse caso. Suponha que o editor seja escrito como um programa com duas *threads*. Uma delas interage com o usuário e a outra lida com a reformatação em segundo plano. Tão logo a frase é apagada da página 1, a thread interativa diz a de reformatação para reformatar o livro inteiro. Enquanto isso, a thread interativa continua a ouvir o teclado e o mouse e responde a comandos simples como rolar a página 1 enquanto a outra thread está trabalhando com afinco no segundo plano. Com um pouco de sorte, a reformatação será concluída antes que o usuário peça para ver a página 600, então ela pode ser exibida instantaneamente. E por quê não acrescentar uma terceira thread? Há editores com capacidade de salvar automaticamente o arquivo inteiro para o disco em intervalos de poucos minutos para proteger o usuário contra a perda de trabalho, caso o programa ou o sistema trave ou falte luz. A terceira thread pode fazer backups de disco sem interferir nas outras duas threads.

Se o programa tivesse apenas uma thread, então sempre que um backup de disco fosse iniciado, comandos do teclado e do mouse seriam ignorados até que o backup tivesse sido concluído. O usuário perceberia isso como um desempenho lento. De outro modo, eventos do teclado e do mouse poderiam interromper o backup do disco, permitindo um bom desempenho, mas isso poderia acarretar um modelo de programação complexo orientado à interrupção. Com três threads, o modelo de programação é muito mais simples: a primeira thread apenas interage com o usuário, a segunda reformata o documento quando solicitado e a terceira escreve os conteúdos da RAM para o disco periodicamente.

Assim, três processos em separado não funcionaria neste exemplo, pois as três tarefas precisariam ser executadas no documento (mesmo espaço de endereçamento de memória). Ao existirem três threads em vez de três processos, elas compartilham de uma memória comum e desse modo têm acesso ao documento que está sendo editado. Com três processos isso seria impossível.

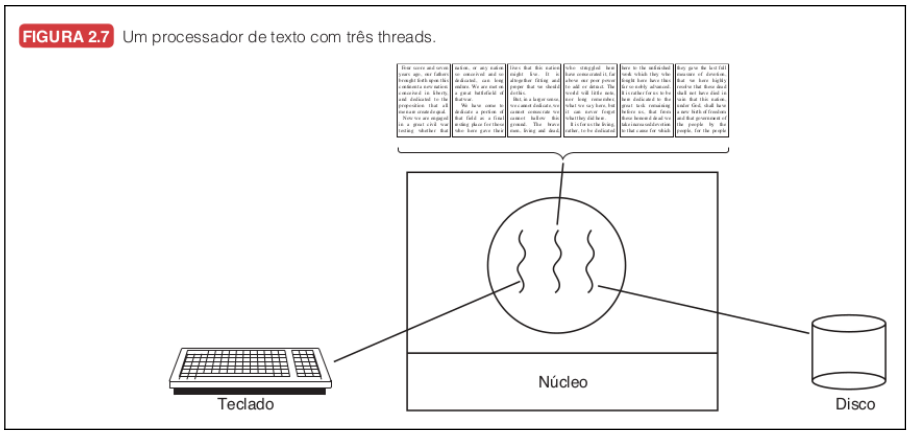


FIGURA 2.7 Um processador de texto com três threads.

2.2.2 O modelo de thread clássico

Baseado em dois conceitos independentes: agrupamento de recursos e execução. Processo é um modo para agrupar recursos relacionados. Ele tem um espaço de endereçamento contendo o código e os dados do programa, além de outros recursos como: arquivos abertos, processos filhos, alarmes pendentes, tratadores de sinais, informação sobre contabilidade, etc. Ao colocá-los juntos na forma de um processo, eles podem ser gerenciados mais facilmente.

Processo contém uma ou mais linhas (threads) de execução. A thread tem:

1. Contador de programa: controla qual é a próxima instrução a ser executada.
2. Registradores: armazenam suas variáveis de trabalho atuais.
3. Pilha: contém o histórico de execução, com uma estrutura para cada rotina chamada, mas ainda não retornada.

Embora uma thread deva executar em algum processo, a thread e seu processo são conceitos diferentes e podem ser tratados separadamente. Processos são usados para agrupar recursos; threads são as entidades escalonadas para execução na CPU.

O que as threads acrescentam para o modelo de processo é permitir que ocorram múltiplas execuções no mesmo ambiente, com um alto grau de independência. A execução de múltiplas threads em paralelo num processo equivale a ter múltiplos processos executando em paralelo em um computador. No primeiro caso, as threads compartilham um espaço de endereçamento e outros

recursos. No segundo caso, os processos compartilham memórias físicas, discos, impressoras, etc. Como *threads* têm algumas das propriedades dos processos, às vezes elas são chamadas de processos leves. O termo *multithread* é usado para descrever a situação de permitir múltiplos threads no mesmo processo. Como vimos no Capítulo 1, algumas CPUs têm suporte de hardware para *multithread* e permitem que chaveamentos de *threads* aconteçam numa escala de tempo de nanossegundos.

Na Figura 2.11(a) vemos 3 processos. Cada processo tem seu próprio espaço de endereçamento e uma única *thread* de controle. Em comparação, na Figura 2.11(b) vemos um único processo com três *threads* de controle. Embora em ambos os casos tenhamos três *threads*, na Figura 2.11(a) cada uma delas opera em um espaço de endereçamento diferente, enquanto na Figura 2.11(b) todos os três compartilham o mesmo espaço de endereçamento.

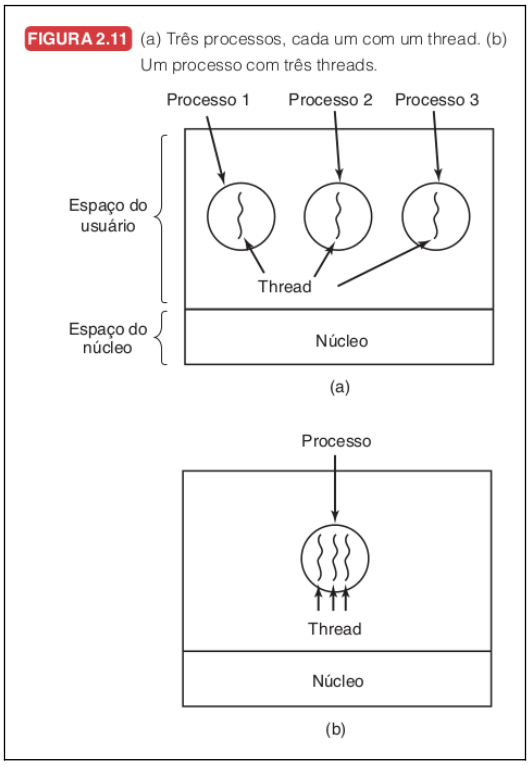


FIGURA 2.11 (a) Três processos, cada um com um thread. (b) Um processo com três threads.

Quando um processo *multithread* é executado em um sistema de CPU única, as *threads* se revezam executando. Ao chavear entre múltiplos processos, o sistema passa a ilusão de processos sequenciais executando em paralelo. O *multithread* funciona da mesma maneira. A CPU chaveia rapidamente entre as *threads*, dando a ilusão de que elas estão executando em paralelo. Num processo limitado pela CPU com três *threads*, elas pareceriam executar em paralelo, cada uma numa CPU com 1/3 da velocidade da CPU real.

Threads diferentes em um processo não são tão independentes quanto processos diferentes. Todas as *threads* têm exatamente o mesmo espaço de endereçamento, compartilhando as mesmas variáveis globais. Tendo em vista que toda *thread* pode acessar todo espaço de endereçamento de memória dentro do espaço de endereçamento do processo, uma *thread* pode ler, escrever, ou mesmo apagar a pilha de outra *thread*. Não há proteção entre threads, porque (1) é impossível e (2) não seria necessário. Ao contrário de processos distintos, que podem ser de usuários diferentes e que podem ser hostis uns com os outros, um processo é sempre propriedade de um único usuário, que presumivelmente criou múltiplas *threads* de maneira que elas possam cooperar, não lutar. Além de

compartilhar um espaço de endereçamento, todas as *threads* podem compartilhar o mesmo conjunto de arquivos abertos, processos filhos, alarmes e sinais, como mostrado na Figura 2.12. Assim, a organização da Figura 2.11(a) seria usada quando os 3 processos forem essencialmente não relacionados, enquanto a Figura 2.11(b) seria apropriada quando as 3 threads fizerem parte do mesmo trabalho e estiverem cooperando umas com as outras de maneira ativa e próxima.

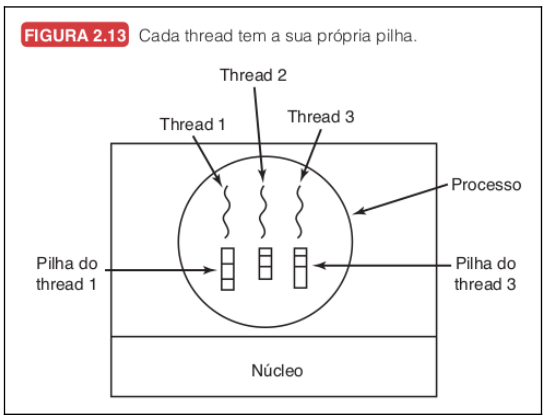
Na Figura 2.12, os itens na primeira coluna são propriedades de processos, não de *threads*. Se uma *thread* abre um arquivo, este fica visível as outras *threads* no processo e elas podem ler e escrever nele. A *thread* não é a unidade de gerenciamento de recursos. Se cada *thread* tivesse o seu próprio espaço de endereçamento, arquivos abertos, alarmes pendentes, etc, seria um processo em separado. A ideia de *thread* esta relacionada a capacidade de execução múltipla de um conjunto compartilhado de recursos, para desempenhar alguma tarefa.

FIGURA 2.12 A primeira coluna lista alguns itens compartilhados por todos os threads em um processo. A segunda lista alguns itens específicos a cada thread.

Itens por processo	Itens por thread
Espaço de endereçamento	Contador de programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos filhos	Estado
Alarmes pendentes	
Sinais e tratadores de sinais	
Informação de contabilidade	

Como um processo tradicional (isto é, um processo com apenas uma *thread*), uma *thread* pode estar em qualquer um desses estados: em execução, bloqueado ou pronto. Uma *thread* em execução tem a CPU naquele momento e está ativo. Em comparação, uma *thread* bloqueada está esperando por algum evento para desbloqueá-la. Por exemplo, quando uma *thread* realiza uma chamada de sistema para ler do teclado, ela está bloqueada até que uma entrada seja digitada. Uma *thread* pode bloquear esperando por algum evento externo acontecer ou por alguma outra *thread* para desbloqueá-la. Uma *thread* pronta está programada para ser executada e será - tão logo chegue a sua vez. As transições entre estados de *thread* são as mesmas que aquelas entre estados de processos e estão ilustradas na Figura 2.2.

Cada *thread* tem a sua própria pilha, como ilustrado na Figura 2.13. Cada pilha da thread contém uma estrutura para cada rotina chamada, mas ainda não retornada. Essa estrutura contém as variáveis locais da rotina e o endereço de retorno para usar quando a chamada de rotina for encerrada. Cada *thread* geralmente chamará rotinas diferentes e desse modo terá uma história de execução diferente. Essa é a razão pela qual cada *thread* precisa da sua própria pilha.



Quando o *multithreading* está presente, os processos normalmente começam com uma única *thread* presente. Essa *thread* tem a capacidade de criar novas, chamando uma rotina de biblioteca como thread_create. Um parâmetro para thread_create especifica o nome de uma rotina para a nova thread executar. Não é necessário (ou mesmo possível) especificar algo sobre o espaço de endereçamento da nova *thread*, tendo em vista que ela automaticamente é executada no espaço de endereçamento da *thread* em criação. Às vezes, threads são hierárquicas, com uma relação pai-filho, mas muitas vezes não existe uma relação dessa natureza, e todas as *threads* são iguais. Com ou sem uma relação hierárquica, normalmente é devolvida à *thread* em criação um identificador de *thread* que nomeia a nova *thread*.

Quando uma thread tiver terminado o trabalho, pode concluir sua execução chamando uma rotina de biblioteca, como thread_exit. Ela então desaparece e não é mais escalonável. Em alguns sistemas, uma thread pode esperar pela saída de uma thread (específica) chamando uma rotina, por exemplo, thread_join. Essa rotina bloqueia a thread que executou a chamada até que uma thread (específica) tenha terminado. Nesse sentido, a criação e a conclusão de threads é muito semelhante à criação e ao término de processos, com mais ou menos as mesmas opções.

Outra chamada de thread comum é thread_yield, que permite que uma thread abra mão voluntariamente da CPU para deixar outra thread ser executada. Uma chamada dessas é importante porque não há uma interrupção de relógio para realmente forçar a multiprogramação como há com os processos. Desse modo, é importante que as threads sejam educadas e voluntariamente entreguem a CPU de tempos em tempos para dar as outras threads uma chance de serem executadas. Outras chamadas permitem que uma thread espere por outra para concluir algum trabalho.

2.2.3 Threads POSIX

Para possibilitar que se escrevam programas com threads, o IEEE definiu um padrão para threads no padrão IEEE 1003.1c. O pacote de threads que ele define é chamado Pthreads. A maioria dos sistemas UNIX dá suporte a ele.

Cada thread tem um identificador, um conjunto de registradores (incluindo o contador de programa), e um conjunto de atributos, que são armazenados em uma estrutura. Os atributos incluem tamanho da pilha, parâmetros de escalonamento e outros itens necessários para usar a thread.

FIGURA 2.14 Algumas das chamadas de função do Pthreads.

Chamada de thread	Descrição
Pthread_create	Cria um novo thread
Pthread_exit	Conclui a chamada de thread
Pthread_join	Espera que um thread específico seja abandonado
Pthread_yield	Libera a CPU para que outro thread seja executado
Pthread_attr_init	Cria e inicializa uma estrutura de atributos do thread
Pthread_attr_destroy	Remove uma estrutura de atributos do thread

Uma nova thread é criada usando a chamada pthread_create. O identificador de uma *thread* recentemente criada é retornado como o valor da função. O identificador da thread desempenha o papel do PID (número de processo). Quando uma *thread* tiver acabado o trabalho para o qual foi designada, ela pode terminar chamando pthread_exit. Essa chamada para a *thread* libera sua pilha.

Muitas vezes, uma *thread* precisa esperar outra terminar seu trabalho. A que está esperando chama pthread_join para esperar outra *thread* terminar. O identificador da *thread* esperada é dado como parâmetro.

Às vezes acontece de uma *thread* estar em execução, mas já foi executada por tempo suficiente e concede a outra *thread* a chance de ser executada. Ela pode atingir essa meta chamando pthread_yield. Essa chamada não existe para processos, pois o pressuposto aqui é que os processos são altamente competitivos e cada um quer o tempo de CPU que conseguir obter. No entanto, já que as *threads* de um processo estão trabalhando juntas e seu código é invariavelmente escrito pelo mesmo programador, às vezes o programador quer que elas se deem outra chance.

Pthread_attr_init cria a estrutura de atributos associada com uma *thread* e a inicializa com os valores padrão. Esses valores (como a prioridade) podem ser modificados manipulando campos na estrutura de atributos. Por fim, pthread_attr_destroy remove a estrutura de atributos de uma *thread*, liberando a sua memória. Essa chamada não afeta as *threads* que a usam; elas continuam a existir.

FIGURA 2.15 Um exemplo de programa usando threads.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS    10

void *print_hello_world(void *tid)
{
      /* Esta funcao imprime o identificador do thread e sai. */
      printf("Ola mundo. Boas vindas do thread %d\n", tid);
      pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
      /* O programa principal cria 10 threads e sai. */
      pthread_t threads[NUMBER_OF_THREADS];
      int status, i;

      for(i=0; i < NUMBER_OF_THREADS; i++) {
            printf("Metodo Main. Criando thread %d\n", i);
            status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

            if (status != 0) {
                  printf("Oops. pthread_create returned error code %d\n", status);
                  exit(-1);
            }
      }
      exit(NULL);
}
```

2.2.4 Implementando threads no espaço do usuário

Há dois lugares principais para implementar threads: no espaço do usuário e no núcleo. O primeiro método é colocar o pacote de threads inteiramente no espaço do usuário. O núcleo não sabe nada a respeito deles. Com essa abordagem, threads são implementados por uma biblioteca. Todas essas implementações têm a mesma estrutura geral, ilustrada na Figura 2.16(a).

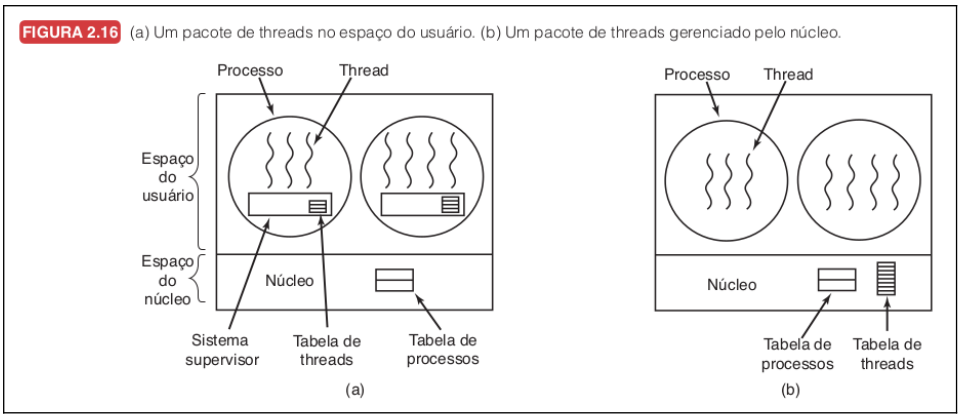


FIGURA 2.16 (a) Um pacote de threads no espaço do usuário. (b) Um pacote de threads gerenciado pelo núcleo.

Quando as threads são gerenciadas no espaço do usuário, cada processo precisa da sua própria tabela de threads privada para controlá-los naquele processo. Ela é análoga à tabela de processo do núcleo, exceto por controlar apenas as propriedades de cada thread, como o contador de programa, o ponteiro de pilha, registradores e estado. A tabela de threads é gerenciada pelo sistema de tempo de execução. Quando um thread vai para o estado pronto ou bloqueado, a informação necessária para reiniciá-lo é armazenada na tabela de threads, exatamente da mesma maneira que o núcleo armazena informações sobre processos na tabela de processos. Lidar, desta forma, com threads é pelo menos uma ordem de magnitude — talvez mais — mais rápida do que desviar o controle para o núcleo, além de ser um forte argumento a favor de pacotes de threads de usuário.

No entanto, há uma diferença fundamental com os processos. Quando um thread decide parar de executar — por exemplo, quando ele chama thread_yield — o código de thread_yield pode salvar as informações do thread na própria tabela de thread. Além disso, ele pode então chamar o escalonador de threads para pegar outro thread para executar. A rotina que salva o estado do thread e o escalonador são apenas rotinas locais, de maneira que invocá-las é algo muito mais eficiente do que fazer uma chamada de núcleo. Entre outras coisas, nenhuma armadilha (trap) é necessária, nenhum chaveamento de contexto é necessário, a cache de memória não precisa ser esvaziada e assim por diante. Isso torna o escalonamento de thread muito rápido.

Threads de usuário também têm outras vantagens. Eles permitem que cada processo tenha seu próprio algoritmo de escalonamento customizado. Eles também escalam melhor, já que threads de núcleo sempre exigem algum espaço de tabela e de pilha no núcleo, o que pode ser um problema se houver um número muito grande de threads.

2.2.5 Implementando threads no núcleo

Agora vamos considerar que o núcleo saiba sobre as threads e as gerencie. Não é necessário um sistema de tempo de execução em cada um, como mostrado na Figura 2.16(a). Também não há uma tabela de thread em cada processo. Em vez disso, o núcleo tem uma tabela que controla todos os threads no sistema - Figura 2.16(b). Quando uma thread quer criar uma nova ou destruir uma

existente, ela faz uma chamada de núcleo, que então faz a criação ou a destruição atualizando a tabela de threads do núcleo.

A tabela de threads do núcleo contém os registradores, estado e outras informações de cada thread. A informação é a mesma com as threads de usuário, mas agora mantidas no núcleo em vez de no espaço do usuário (dentro do sistema de tempo de execução). Essa informação é um subconjunto das informações que os núcleos tradicionais mantêm a respeito dos seus processos de thread único, isto é, o estado de processo. Além disso, o núcleo também mantém a tabela de processos tradicional para controlar os processos.

Todas as chamadas que poderiam bloquear uma thread são implementadas como chamadas de sistema, a um custo consideravelmente maior do que uma chamada para uma rotina de sistema de tempo de execução. Quando uma thread é bloqueada, o núcleo tem a opção de executar outra thread do mesmo processo (se uma estiver pronta) ou alguma thread de um processo diferente. Com threads de usuário, o sistema de tempo de execução segue executando threads a partir do seu próprio processo até o núcleo assumir a CPU dele (ou não houver mais threads prontos para serem executados).

A principal desvantagem das threads implementadas no núcleo é que o custo de uma chamada de sistema é substancial, então se as operações de thread (criação, término etc.) forem frequentes, ocorrerá uma sobrecarga muito maior.

2.2.6 Implementações híbridas

Há maneiras de combinar as vantagens de threads de usuário com threads de núcleo. Uma delas é usar threads de núcleo e então multiplexar as threads de usuário, como mostrado na Figura 2.17. Esse modelo proporciona o máximo em flexibilidade. Com essa abordagem, o núcleo está consciente apenas das threads de núcleo e as escalona. Algumas dessas threads podem ter, em cima delas, múltiplas threads de usuário multiplexadas, as quais são criadas, destruídas e escalonadas exatamente como threads de usuário, num processo executado em um sistema operacional. Nesse modelo, cada thread de núcleo tem algum conjunto de threads de usuário que se revezam para usá-la.

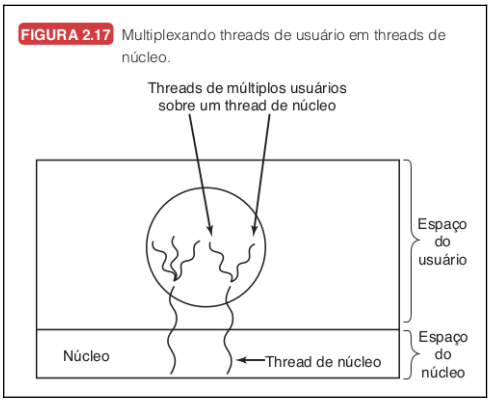


FIGURA 2.17 Multiplexando threads de usuário em threads de núcleo.

2.2.7 Ativações pelo escalonador

Embora threads de núcleo sejam melhores do que threads de usuário em certos aspectos-chave, eles são também indiscutivelmente mais lentas. Como consequência, pesquisadores procuraram maneiras de melhorar a situação sem abrir mão de suas boas propriedades. A seguir

descreveremos uma abordagem desenvolvida por Anderson et al. (1992), chamada ativações pelo escalonador.

A meta do trabalho da ativação pelo escalonador é imitar a funcionalidade das threads de núcleo, mas com melhor desempenho e maior flexibilidade normalmente associados aos pacotes de threads implementadas no espaço do usuário. A eficiência é alcançada evitando-se transições desnecessárias entre espaço do usuário e do núcleo. Se uma thread é bloqueada esperando por outra para fazer algo, por exemplo, não há razão para envolver o núcleo, poupando assim a sobrecarga da transição núcleo-usuário. O sistema de tempo de execução pode bloquear a thread de sincronização e escalonar sozinho uma nova. A ideia básica para o funcionamento desse esquema é a seguinte: quando sabe que uma thread foi bloqueada (por exemplo, tendo executado uma chamada de sistema bloqueante ou causado uma falta de página), o núcleo notifica o sistema de tempo de execução do processo, passando como parâmetros na pilha o número da thread em questão e uma descrição do evento que ocorreu. A notificação acontece quando o núcleo ativa o sistema de tempo de execução em um endereço inicial conhecido, de maneira mais ou menos análoga a um sinal em UNIX. Esse mecanismo é chamado upcall.

2.2.8 Threads pop-up

Threads são úteis em sistemas distribuídos. Um exemplo importante é como mensagens que chegam — por exemplo, requisições de serviços — são tratadas. A abordagem tradicional é ter um processo ou thread que esteja bloqueado em uma chamada de sistema receive esperando pela mensagem que chega. Quando uma mensagem chega, ela é aceita, aberta, seu conteúdo examinado e processada.

No entanto, uma abordagem completamente diferente também é possível, na qual a chegada de uma mensagem faz o sistema criar uma nova thread para lidar com a mensagem. Essa thread é chamado de thread pop-up e está ilustrado na Figura 2.18. Uma vantagem fundamental de threads pop-up é que como são novas, elas não têm história alguma, e portanto nada que precise ser restaurado nos seus registradores, pilha, etc. Cada uma é iniciada de forma idêntica a todas as outras. Isso possibilita a criação de tais threads rapidamente. A thread nova recebe a mensagem que chega para processar. O resultado da utilização de threads pop-up é que a latência entre a chegada da mensagem e o começo do processamento pode ser encurtada.

Executar uma thread pop-up no espaço núcleo normalmente é mais fácil e mais rápido do que colocá-la no espaço do usuário. Também, uma thread pop-up no espaço núcleo consegue facilmente acessar todas as tabelas do núcleo e os dispositivos de E/S, que podem ser necessários para o processamento de interrupções. Por outro lado, uma thread de núcleo com erros pode causar mais danos que uma de usuário com erros. Por exemplo, se ela for executada por tempo demais e não liberar a CPU, dados que chegam podem ser perdidos.

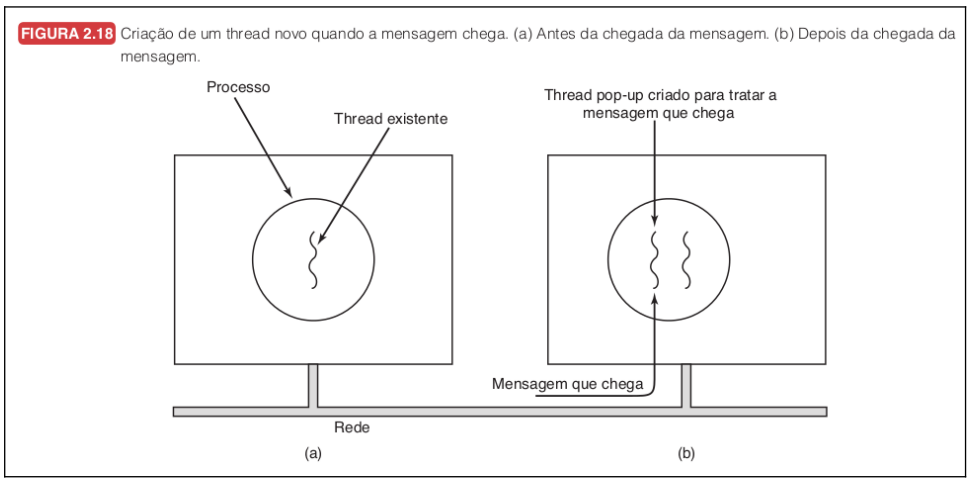


FIGURA 2.18 Criação de um thread novo quando a mensagem chega. (a) Antes da chegada da mensagem. (b) Depois da chegada da mensagem.

2.3 Comunicação entre processos

Processos quase sempre precisam comunicar-se com outros processos, de preferência de uma maneira bem estruturada sem usar interrupções. Há 3 questões envolvidas: (1) um processo pode passar informações para outro; (2) certificar-se de que dois ou mais processos não se atrapalhem; e (3) sequenciamento adequado quando dependências estão presentes.

As mesmas questões aplicam-se às threads. Como compartilham o mesmo espaço de endereçamento, é mais fácil passar informação entre elas. Threads em espaços de endereçamento diferentes que precisam comunicar-se, são questões relativas à comunicação entre processos.

2.3.1 Condições de corrida (de concorrência)

Processos comunicam-se através de uma área de armazenamento comum. Essa área pode estar na memória principal ou um arquivo compartilhado. Mas, é possível que haja conflito. Esse conflito é chamado de condição de corrida, caracterizado pela situação onde dois ou mais processos acessam recursos compartilhados de forma concorrente. Há uma corrida pelo recurso.

(1) Exemplo matemático simples:

a = d + c

x = **a** + y

Neste exemplo a variável “a” é compartilhada e “x” depende de “a”. Se estes eventos estão em processos/threads separadas, é necessário sincronizá-los.

(2) Exemplo edição de célula:

Tentativa simultânea de execução de uma célula, numa planilha compartilhada.

(3) Exemplo em companhia aérea:

Dois processos em um sistema de reserva de uma companhia aérea, cada um tentando ficar com o último assento em um avião para um cliente diferente.

(4) Exemplo de um *spool* de impressão:

Há duas variáveis compartilhadas: *out*, que aponta para o próximo arquivo a ser impresso; e *in*, que aponta para a próxima vaga livre no diretório. De maneira mais ou menos simultânea, os processos A e B decidem que querem colocar um arquivo na fila para impressão. O processo A lê *in*

e armazena o valor 7 em uma **variável local** chamada ***next_free_slot***. Logo em seguida uma interrupção de relógio ocorre e a CPU decide que o processo A executou por tempo suficiente, então, ele troca para o processo B. O processo B também lê *in* e recebe um 7. Ele também o armazena em sua variável local *next_free_slot*. Nesse instante, ambos os processos acreditam que a próxima vaga disponível é 7. O processo B agora continua a executar. Ele armazena o nome do seu arquivo na vaga 7 e atualiza *in* para ser 8. Então, ele segue em frente para fazer outras coisas.

Por fim, o processo A executa novamente, começando do ponto onde ele parou. Ele olha para *next_free_slot*, encontra um 7 ali e escreve seu nome de arquivo na vaga 7, apagando o nome que o processo B recém colocou ali. Então, calcula *next_free_slot* + 1, que é 8, e configura *in* para 8. O diretório de *spool* está agora internamente consistente, então o *daemon* de impressão não observará nada errado, mas o processo B jamais receberá qualquer saída. O usuário B ficará em torno da impressora por anos, aguardando esperançoso por uma saída que nunca virá.

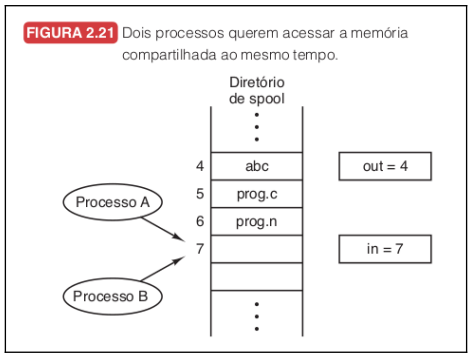


FIGURA 2.21 Dois processos querem acessar a memória compartilhada ao mesmo tempo.

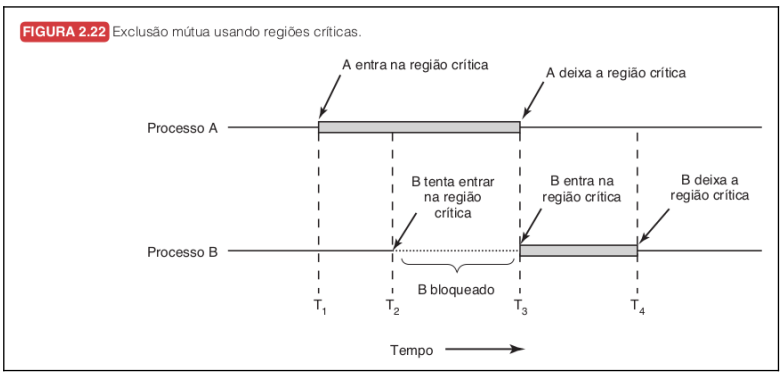
O cenário desejável é que a resolução dos conflitos relacionados às condições de corrida, sejam tratados de maneira bem estruturada para evitar interrupção.

2.3.2 Regiões críticas

Como evitar condições de corrida? Proibir mais de um processo de ler e escrever os dados compartilhados ao mesmo tempo. Ou seja, é necessário que haja uma **exclusão mútua**, isto é, alguma maneira de garantir que, se um processo está usando um arquivo ou variável compartilhados, os outros serão impedidos de realizar o mesmo. A dificuldade mencionada no exemplo 4, ocorreu porque o processo B usou a "vaga 7" antes do processo A ter terminado a execução. Essa parte do programa onde a memória compartilhada é acessada é chamada de região crítica ou seção crítica. Arranjar as coisas de maneira que jamais dois processos estejam em suas regiões críticas ao mesmo tempo, poder evitar corridas (concorrências).

Embora essa exigência evite as condições de corrida, ela não é suficiente para garantir que processos em paralelo cooperem de modo correto e eficiente usando dados compartilhados. Quatro requisitos precisam ser atendidos para solucionar a questão:

1. Dois processos jamais podem estar simultaneamente dentro de uma região crítica.
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 processo.
4. Nenhum processo deve ser obrigado a esperar eternamente para entrar numa região crítica.



2.3.3 Exclusão mútua com espera ocupada

Maneiras de garantir que enquanto um processo está sendo executado na memória compartilhada (em sua região crítica), nenhum outro processo entrará nesta região crítica para causar problemas. Em outras palavras, veremos quais são as 5 soluções existentes para realizar a exclusão mútua: espera ocupada; dormir e acordar; semáforos; monitores; e troca de mensagem. Nesta Seção 2.3.3 serão apresentados as diferentes técnicas da solução envolvendo a espera ocupada.

2.3.3.1 Desabilitando interrupções

Num sistema de processador único, a solução mais simples é fazer com que cada processo desabilite todas as interrupções logo após entrar em sua região crítica e as reabilitar um momento antes de sair dessa região crítica. Com as interrupções desabilitadas, nenhuma interrupção de relógio poderá ocorrer. Afinal de contas, a CPU só é chaveada de processo em processo em consequência de uma interrupção de relógio ou outra, e com as interrupções desligadas, a CPU não será chaveada para outro processo. Então, assim que um processo tiver desabilitado as interrupções, ele poderá examinar e atualizar a memória compartilhada sem medo de que qualquer outro processo interfira.

Em geral, essa abordagem é pouco atraente, pois não é prudente dar aos processos de usuário o poder de desligar interrupções. E se um deles desligasse uma interrupção e nunca mais a ligasse de volta? Isso poderia ser o fim do sistema. Além disso, se o sistema é um multiprocessador (com duas ou mais CPUs), desabilitar interrupções afeta somente a CPU que executou a instrução *disable*. As outras continuarão executando e podem acessar a memória compartilhada.

A conclusão é: desabilitar interrupções é muitas vezes uma técnica útil no núcleo do sistema operacional, mas não é apropriada como um mecanismo de exclusão mútua para processos de usuário.

A possibilidade de alcançar a exclusão mútua desabilitando interrupções — mesmo dentro do núcleo — está se tornando menor a cada dia por causa do número cada vez maior de chips multinúcleo até em PCs populares. Em um sistema multinúcleo (isto é, sistema de multiprocessador) desabilitar as interrupções de uma CPU não evita que outras CPUs interfiram nas operações que a primeira está realizando. Em consequência, esquemas mais sofisticados são necessários.

2.3.3.2 Variáveis do tipo trava

Uma solução seria considerar a presença de uma única variável (de trava) compartilhada, inicialmente 0. Quando um processo quer entrar em sua região crítica, ele primeiro testa a trava. Se a trava é 0, o processo entra na região crítica e configura a trava para 1. Se a trava já é 1, o processo apenas espera até que ela se torne 0. Desse modo, um **0 significa que nenhum processo está na região crítica, e um 1 significa que algum processo está em sua região crítica**.

Suponha que um processo lê a trava e vê que ela é 0. Antes que ele possa configurar a trava para 1, outro processo está escalonado, executa e configura a trava para 1. Quando o primeiro processo executa de novo, ele também configurará a trava para 1 (sobrescrevendo a variável de trava), e dois processos estarão nas suas regiões críticas ao mesmo tempo.

2.3.3.3 Alternância explícita

No código da Figura 2.23 e na Tabela 1, observa-se que a variável do tipo inteiro *turn*, inicialmente 0, serve para **controlar de quem é a vez de entrar na região crítica** e examinar ou atualizar a memória compartilhada. Inicialmente, o processo 0 inspeciona *turn*, descobre que ele é 0 e entra na sua região crítica. O processo 1 também encontra lá o valor 0 e, portanto, espera em um laço fechado testando continuamente *turn* para ver quando ele vira 1. **Testar continuamente uma variável até que algum valor apareça é chamado de espera ocupada**. Em geral, ela **deve ser evitada, já que desperdiça tempo da CPU**. Apenas quando há uma expectativa razoável de que a espera será curta, a espera ocupada é usada. Uma trava que usa a espera ocupada é chamada de trava giratória (*spin lock*).

O processo 0 configura *turn* para 1 e deixa a região crítica, a fim de permitir que o processo 1 entre em sua região crítica. Suponha que o processo 1 termine sua região rapidamente, de modo que ambos os processos estejam em suas regiões não-críticas, com *turn* configurado para 0. Agora, o processo 0 executa todo seu laço rapidamente, configura *turn* para 1 e deixa sua região crítica. Nesse ponto, *turn* é 1 e ambos os processos estão sendo executados em suas regiões não-críticas.

Tabela 1: Resumo do fluxo do processo até a ocorrência da espera ocupada.

Turn	Processo	Região Crítica	Região Não-Crítica	Observação
0	0	Sim	Não	
1	0	Não	Sim	
1	1	Sim	Não	Suponha que o processo 1 tenha executado rapidamente.
0	1	Não	Sim	Ambos os processos estão na região não-crítica.
0	0	Sim	Não	Suponha que o processo 0 tenha executado rapidamente.
1	0	Não	Sim	Ambos os processos estão na região não-crítica.
Neste ponto, se o Processo 0 terminar sua execução na região não-crítica de forma mais rápida que o processo 1 e tiver que voltar para a região crítica, ele não consegue porque turn é 1 e o processo 1 ainda está ocupado na região não-crítica.				

De repente, o processo 0 termina sua região não-crítica e volta para o topo do seu laço. Infelizmente, não lhe é permitido entrar em sua região crítica agora, pois *turn* é 1 e o processo 1 está ocupado com sua região não-crítica. Ele espera em seu laço *while* até que o processo 1 configura *turn* para 0. Ou seja, **chavear a vez não é uma boa ideia quando um dos processos é muito mais lento que o outro**.

Essa situação viola a condição 3 estabelecida anteriormente: **o processo 0 está sendo bloqueado por um que não está em sua região crítica**.

FIGURA 2.23 Uma solução proposta para o problema da região crítica. (a) Processo 0. (b) Processo 1. Em ambos os casos, certifique-se de observar os pontos e vírgulas concluindo os comandos while.

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

(a)

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

(b)

2.3.3.4 Solução de Peterson

Em 1981, G. L. Peterson descobriu um modelo de realização da exclusão mútua. O algoritmo de Peterson é mostrado na Figura 2.24.

Antes de usar as variáveis compartilhadas (isto é, antes de entrar na região crítica), cada processo chama *enter_region* com seu próprio número de processo, 0 ou 1, como parâmetro. Essa chamada fará com que ele espere, se necessário, até que seja seguro entrar. Após haver terminado com as variáveis compartilhadas, o processo chama *leave_region* para indicar que ele terminou e para permitir que outros processos entrem, se assim desejarem.

Inicialmente, nenhum processo está na sua região crítica. Agora o processo 0 chama *enter_region*. Ele indica o seu interesse alterando o valor de seu elemento de arranjo e alterando *turn* para 0. Como o processo 1 não está interessado, *enter_region* retorna imediatamente. Se o processo 1 fizer agora uma chamada para *enter_region*, ele esperará ali até que *interested[0]* mude para *FALSE*. Um evento que acontece apenas quando o processo 0 chamar *leave_region* para deixar a região crítica.

Agora, considere o caso em que ambos os processos chamam *enter_region* quase simultaneamente. Ambos armazenarão seu número de processo em *turn*. O último a armazenar é o que conta; o primeiro é sobrescrito e perdido. Suponha que o processo 1 armazene por último, então *turn* é 1.

FIGURA 2.24 A solução de Peterson para realizar a exclusão mútua.

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

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

void enter_region(int process);         /* processo e 0 ou 1 */
{
    int other;                          /* numero do outro processo */

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

void leave_region(int process)          /* processo: quem esta saindo */
{
    interested[process] = FALSE;        /* indica a saida da regiao critica */
}
```

2.3.3.4.1 A instrução TSL

A instrução TSL (da linguagem *Assembly*) é uma das formas de implementação do modelo de Peterson. Esta proposta exige a ajuda do *hardware*. Computadores com múltiplos processadores, têm uma instrução como TSL RX,LOCK (Test and Set Lock — teste e configuração de trava) que lê o conteúdo da variável *lock* da memória para o registrador RX. Nenhum outro processador pode acessar a variável na memória até que a instrução tenha terminado. A CPU executando a instrução TSL impede o acesso ao barramento de memória para proibir que outras CPUs acessem a memória até ela terminar.

É importante observar que impedir o barramento de memória é diferente de desabilitar interrupções. Desabilitar interrupções e então realizar a leitura de uma variável na memória seguida pela escrita não evita que um segundo processador no barramento acesse a variável entre esses eventos de leitura e a escrita. Na realidade, desabilitar interrupções no processador 1 não exerce efeito algum sobre o processador 2. A única maneira de manter o processador 2 fora da memória até o processador 1 ter terminado é travar o barramento, o que exige um equipamento de hardware especial (basicamente, uma linha de barramento que assegura que o barramento está travado e indisponível para outros processadores - fora aquele que o travar).

Como essa instrução pode ser usada para evitar que dois processos entrem simultaneamente em suas regiões críticas? A solução é dada na Figura 2.25. Nela, uma sub-rotina de quatro instruções é apresentada numa linguagem de montagem. A primeira instrução copia o valor de *lock* para o registrador e, então, configura *lock* para 1. Assim, o valor é comparado a 0. Se não for 0, o programa volta para o início e o testa novamente. Cedo ou tarde ele se tornará 0 (quando o processo atualmente em sua região crítica tiver terminado sua execução), e a sub-rotina retornar, com a trava configurada. Para destravá-la o programa armazena um 0 em *lock*.

Antes de entrar em sua região crítica, um processo chama *enter_region*, que fica em espera ocupada até a trava estar livre; então ele adquire a trava e entra na região crítica. Após deixar a região crítica, o processo chama *leave_region*, que armazena um 0 em *lock*.

Uma instrução alternativa para TSL é XCHG, que troca os conteúdos de duas posições atomicamente; por exemplo, um registrador e uma variável de memória. O código é mostrado na

Figura 2.26, e, como podemos ver, é essencialmente o mesmo que a solução com TSL. Todas as CPUs Intel x86 usam a instrução XCHG para a sincronização de baixo nível.

FIGURA 2.25 Entrando e saindo de uma região crítica usando a instrução TSL.

```
enter_region:
        TSL REGISTER,LOCK       | copia lock para o registrador e poe lock em 1
        CMP REGISTER,#0         | lock valia zero?
        JNE enter_region        | se fosse diferente de zero, lock estaria ligado; portanto,
                                  continue no laco de repeticao
        RET                     | retorna a quem chamou; entrou na regiao critica


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

FIGURA 2.26 Entrando e saindo de uma região crítica usando a instrução XCHG.

```
enter_region:
        MOVE REGISTER,#1        | insira 1 no registrador
        XCHG REGISTER,LOCK      | substitua os conteudos do registrador e a variacao de lock
        CMP REGISTER,#0         | lock valia zero?
        JNE enter_region        | se fosse diferente de zero, lock estaria ligado; portanto
                                  continue no laco de repeticao
        RET                     | retorna a quem chamou; entrou na regiao critica


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

As soluções de Peterson têm o defeito de necessitar da espera ocupada. Ou seja, quando um processo/thread quer entrar em sua região crítica, ele confere para ver se a entrada é permitida. Se não for, o processo aguardará em *loop*, desperdiçando tempo da CPU (ciclos de processador).

Além disso, pode haver o problema da inversão de prioridade, que acontece quando um processo H de mais alta prioridade está no estado de "pronto" e precisa esperar pela CPU – que está alocada para um processo L de baixa prioridade. Caso haja um ou mais processos de prioridade média M, H precisa esperar ainda mais para usar o recurso CPU. Isso gera um impasse (*deadlock*), que afeta, principalmente, o exercício da prioridade daqueles processos com prioridade mais elevada. Tal situação, pode ser evitada pela implementação do protocolo de herança de prioridade. De acordo com este protocolo, todos os processos que acessam recursos necessários para um processo de maior prioridade herdam a prioridade mais alta até terminarem o uso dos recursos em questão. Quando terminam, suas prioridades voltam aos valores originais. No exemplo acima, um protocolo de herança de prioridade permitiria ao processo L herdar temporariamente a prioridade do processo H, evitando assim, que o processo M antecipasse a sua execução. Quando o processo L terminasse de usar o recurso S (CPU, neste caso), ele abandonaria a prioridade herdada de H e assumiria a prioridade original. Como o recurso S estaria agora disponível, o processo H — e não M — seria executado em seguida (SILBERSCHATZ, et al., 2018). O problema da inversão de prioridade ocorreu com a *Mars Pathfinder*, uma sonda veicular robótica construída pela NASA que pousou em Marte em 1997.

2.3.4 Dormir e acordar

Há primitivas de comunicação capazes de bloquear processos para impedir o desperdício de tempo da CPU quando eles não têm autorização para entrar nas suas regiões críticas. *Sleep* é uma chamada de sistema que faz com que o próprio processo que a chamou bloqueie, até que outro processo o desperte. A chamada *wakeup* tem um parâmetro, o processo a ser desperto.

Tabela 2: Simulação: http://lasdpc.icmc.usp.br/~ssc640/grad/bcc2015/grupoa12/simulation.html

FIGURA 2.27 O problema do produtor-consumidor com uma condição de corrida fatal.

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

void producer(void)
{
     int item;

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


void consumer(void)
{
     int item;

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

2.3.4.1 O problema do produtor-consumidor

Também conhecido como o problema do *buffer* limitado, no qual dois processos compartilham um *buffer* de tamanho fixo. O processo produtor, insere itens no *buffer*, e o processo consumidor retira itens dele.

O problema surge quando o produtor quer colocar um item novo no buffer, mas ele já está cheio. A solução é o produtor ir dormir, para ser desperto quando o consumidor tiver removido, pelo menos, um item. De modo similar, se o consumidor quer remover um item do *buffer* e vê que este está vazio, ele vai dormir até o produtor colocar algo no *buffer* e despertá-lo. É relevante ressaltar que apenas o próprio processo detém a prerrogativa de colocar-se para dormir. No entanto, ele não pode ser desperto por ele mesmo. Apenas um outro processo detém a prerrogativa de acordá-lo.

Essa abordagem leva aos mesmos tipos de condições de corrida que vimos com o diretório de *spool*. Para controlar o número de itens no *buffer*, precisaremos de uma variável, *count*. Se o número máximo de itens que o *buffer* pode conter é N, o código do produtor primeiro testará para ver se *count* é N. Se ele for, o produtor vai dormir; se não for, o produtor acrescentará um item e incrementará *count*. O código do consumidor é similar: primeiro testar *count* para ver se ele é 0. Se for, vai dormir; se não for, remove um item e decresce o contador.

Agora voltemos às condições da corrida. O *buffer* está vazio e o consumidor acabou de ler *count* para ver se é 0. Nesse instante, o escalonador decide parar de executar o consumidor temporariamente (interrupção) e começar a executar o produtor. O produtor insere um item no *buffer*, incrementa *count* e nota que ele agora está em 1. Ponderando que *count* era apenas 0, e assim o consumidor deve estar dormindo, o produtor chama *wakeup* para despertar o consumidor. Infelizmente, o consumidor ainda não está logicamente dormindo, então o sinal (chamada de sistema) de despertar é perdido. Quando o consumidor executa em seguida, ele testará o valor de *count* que ele leu antes, descobrirá que ele é 0 (só que já não é mais!!) e irá dormir. Cedo ou tarde o produtor preencherá todo o *buffer* e vai dormir também. Ambos dormirão para sempre.

A essência do problema é que um chamado de despertar enviado para um processo que (ainda) não está dormindo é perdido. Se não fosse perdido, tudo funcionaria. Uma solução é acrescentar um *bit flag* do sinal de acordar. Quando um sinal de despertar é enviado do processo A (produtor, neste exemplo) para o processo B (consumidor, neste exemplo), essa *flag* recebe 1. Antes que o processo B tente adormecer, se a *flag* for 1, B "percebe" que o processo A tentou acordá-lo. Na sequência, o processo B muda a flag para zero (desliga o *bit*), permanecendo desperto.

Referências complementares sobre o problema do produtor-consumidor e a alternativa de solução com *bit flag*:
https://www.youtube.com/watch?v=qorn1vUac0c
https://www.youtube.com/watch?v=HoSX762N3FE

2.3.5 Semáforos

Em 1965, E. W. Dijkstra sugeriu usar uma variável inteira para contar o número de sinais de acordar salvos para uso futuro. Em sua proposta, um novo tipo de variável, que ele chamava de semáforo, foi introduzido. Um semáforo podia ter o valor 0, indicando que nenhum sinal de despertar fora salvo, ou algum valor positivo se um ou mais sinais de acordar estivessem pendentes.

Dijkstra propôs ter duas operações nos semáforos, hoje normalmente chamadas de *down* e *up* (generalizações de *sleep* e *wakeup*, respectivamente). A operação *down* em um semáforo confere para ver se o valor é maior do que 0. Se for, ele decrementará o valor (isto é, gasta um sinal de acordar armazenado) e apenas continua sua execução na região crítica. Se o valor for 0, o processo é colocado para dormir fora da região crítica - sem completar o *down* para o momento. Conferir o valor, modificá-lo e possivelmente dormir são feitos como uma única ação atômica indivisível. É garantido que uma vez que a operação de semáforo tenha começado, nenhum outro processo pode acessar o semáforo até que a operação tenha sido concluída ou bloqueada (não pode haver interrupção). Essa atomicidade é absolutamente essencial para solucionar problemas de sincronização e evitar condições de corrida. Ações atômicas, nas quais um grupo de operações relacionadas são todas realizadas sem interrupção, são muito relevantes.

A operação *up* incrementa o valor de um determinado semáforo. Se um ou mais processos estiverem dormindo, um deles é acordado para completar seu *down*. A operação de incrementar o semáforo e despertar um processo também é indivisível. Nenhum processo é bloqueado realizando um *up*, assim como nenhum processo é bloqueado realizando um *wakeup*.

2.3.5.1 Solucionando o problema produtor-consumidor usando semáforos

Semáforos solucionam o problema do sinal de acordar perdido, como mostrado na Figura 2.28. Para fazê-los funcionar corretamente, é essencial que eles sejam implementados de uma maneira indivisível. A maneira normal é implementar *up* e *down* como chamadas de sistema, com o sistema operacional desabilitando brevemente todas as interrupções enquanto ele estiver testando o semáforo.

FIGURA 2.28 O problema do produtor-consumidor usando semáforos.

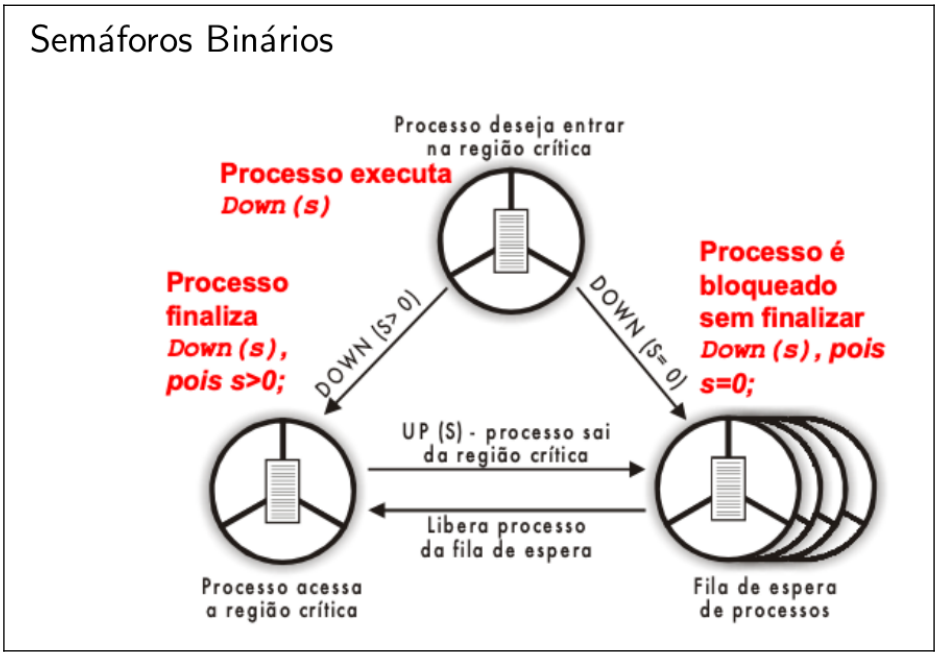
```
#define N 100                                   /* numero de lugares no buffer */
typedef int semaphore;                          /* semaforos sao um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso a regiao critica */
semaphore empty = N;                            /* conta os lugares vazios no buffer */
semaphore full = 0;                             /* conta os lugares preenchidos no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE e a constante 1 */
          item = produce_item( );                /* gera algo para por no buffer */
          down(&empty);                          /* decresce o contador empty */
          down(&mutex);                          /* entra na regiao critica */
          insert_item(item);                     /* poe novo item no buffer */
          up(&mutex);                            /* sai da regiao critica */
          up(&full);                             /* incrementa o contador de lugares preenchidos */
     }
}


void consumer(void)
{
     int item;

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

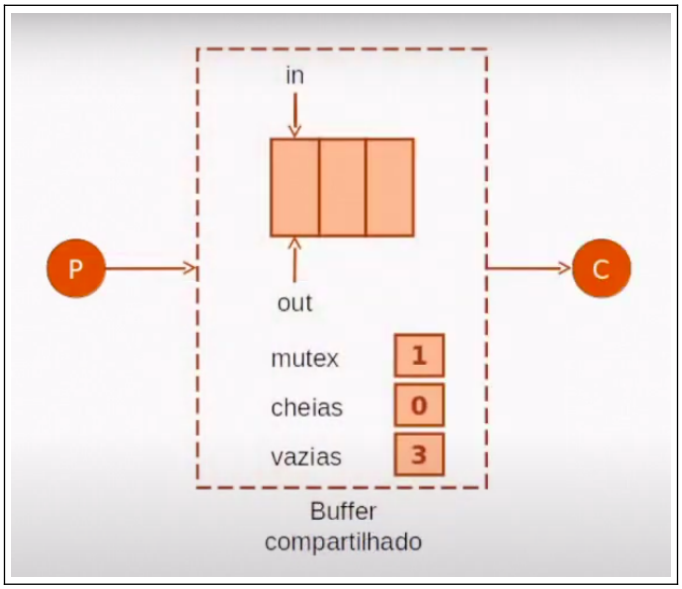
Essa solução usa três semáforos: um chamado *full* para contar o número de vagas que estão cheias, outro chamado *empty* para contar o número de vagas que estão vazias e mais um chamado *mutex* para se certificar de que o produtor e o consumidor não acessem o *buffer* ao mesmo tempo. Inicialmente, *full* é 0, *empty* é igual ao número de vagas no *buffer* e *mutex* é 1. Semáforos que são inicializados para 1 e usados por dois ou mais processos para assegurar que apenas um deles consiga entrar em sua região crítica de cada vez são chamados de semáforos binários. Se cada processo realiza um *down* um pouco antes de entrar em sua região crítica e um *up* logo depois de deixá-la, a exclusão mútua é garantida.

O exemplo da Figura 2.28 lida com semáforos de duas maneiras diferentes. O semáforo *mutex* é usado para exclusão mútua. Ele é projetado para garantir que apenas um único processo de cada vez esteja lendo ou escrevendo no *buffer* e em variáveis associadas. Essa exclusão mútua é necessária para evitar o caos. O outro uso dos semáforos é para a sincronização. Nesse caso, eles asseguram que o produtor pare de executar quando o *buffer* estiver cheio, e que o consumidor pare de executar quando ele estiver vazio.

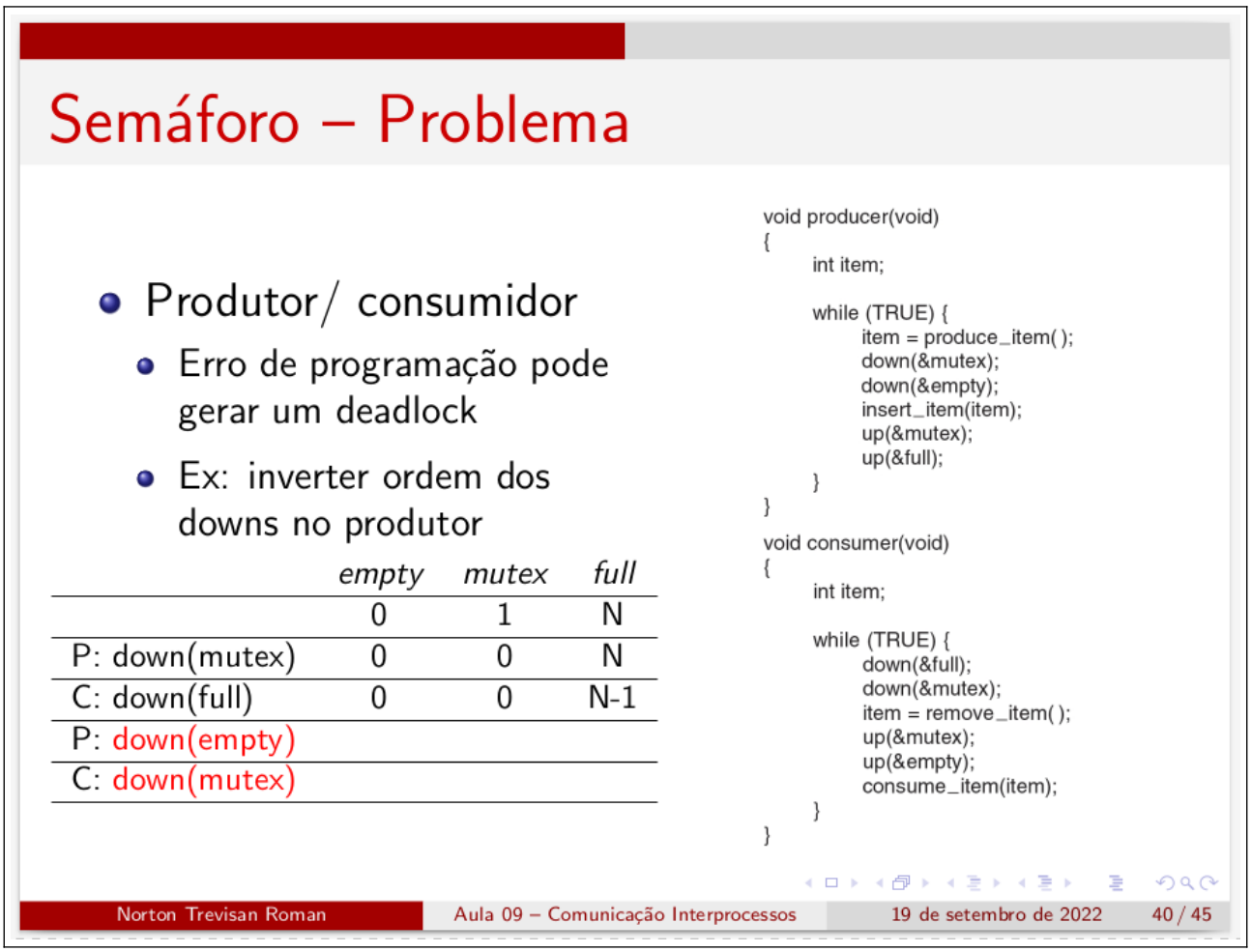
Referências complementares sobre Semáforos:

- Ideia básica: https://youtube.com/clip/UgkxxpNpuwgp78cqGgKbC4YTkQX4-a13tZUs?si=Gm92MgAe1LCCoX8Y





Fonte: simulação simplificada do código da Figura 2.28 em https://www.youtube.com/watch?v=rUHYMZS6Lr4



- **Abordagem completa:** https://www.youtube.com/watch?v=kksOIzamkrY (Neste vídeo há uma análise de 53:37 à 57:22 caracterizando um problema que pode ocorrer em caso de mau uso do semáforo – como exemplo temos a troca das linhas *down(&empty)* *down(&mutex)* no processo produtor da Figura acima). Numa simulação mais realística do código de semáforo da Figura 2.28, *down(&empty)* é interpretado como uma tentativa do processo produtor de adentrar na região crítica. Se o *buffer* já está cheio e, portanto, o *empty* é zero, não há como decrementar ("dar um *down*") no *empty.* Assim, o processo produtor dorme antes de realizar o comando de entrada na região crítica *down(&mutex)*, pois se ele entrasse, não teria o que fazer lá. O problema ocorrerá, caso o programador acidentalmente troque a ordem dos comandos, realizando *down(&mutex)* antes de *down(&empty).* Neste caso, o processo entraria na região crítica e depois tentaria decrementar o "indecrementável", pois o *empty* já seria zero.

2.3.6 Mutexes

Quando a capacidade do semáforo de fazer contagem não é necessária, uma versão simplificada, chamada *mutex*, às vezes é usada. *Mutexes* servem para gerenciar a exclusão mútua de algum recurso ou trecho de código compartilhado. São fáceis e eficientes de implementar, o que os tornam úteis em pacotes de *threads* implementados inteiramente no espaço do usuário.

Um *mutex* é uma variável compartilhada que pode estar em um de dois estados: destravado ou travado. Em consequência, apenas 1 *bit* é necessário para representá-lo, mas na prática muitas vezes um inteiro é usado, com 0 significando destravado e todos os outros valores significando

travado. Duas rotinas são usadas com *mutexes*. Quando uma *thread* (ou processo) precisa de acesso a uma região crítica, ela chama *mutex_lock*. Se o *mutex* estiver destravado naquele momento (significando que a região crítica está disponível), a chamada seguirá e a *thread* que chamou estará livre para entrar na região crítica.

Por outro lado, se o *mutex* já estiver travado, a *thread* que chamou será bloqueada até que a *thread* na região crítica tenha concluído e chame *mutex_unlock*. Se múltiplas *threads* estiverem bloqueadas no *mutex*, uma delas será escolhida ao acaso e liberada para adquirir a trava. Como *mutexes* são muito simples, eles podem ser facilmente implementados no espaço do usuário, desde que uma instrução TSL ou XCHG esteja disponível.

O código de ***mutex_lock* é similar ao código de *enter_region*** da Figura 2.25, mas com uma diferença crucial: **quando *enter_region* falha em entrar na região crítica, ele segue testando a trava repetidamente (espera ocupada)**; por fim, o tempo de CPU termina (pois o temporizador envia um **sinal de interrupção**) e o **processo** que está fora da região crítica é escalonado para executar, pois cedo ou tarde, o **processo** que detém a trava é executado e a libera.

Com ***threads*** (de usuário) a **situação é diferente**, pois **não há um temporizador que pare *threads* que tenham sido executadas por tempo demais**. Em consequência, uma *thread* que tenta adquirir uma trava através da espera ocupada, ficará em um laço para sempre e nunca adquirirá a trava, pois **ela jamais permitirá que outra *thread* execute** e a libere.

É aí que entra a **diferença entre *enter_region* e *mutex_lock***. **Quando *mutex_lock* falha em adquirir uma trava, ele chama *thread_yield* para que a *thread* em execução conceda a CPU para outra *thread***. Em consequência, **não há espera ocupada.** Quando a *thread* executa da vez seguinte, ela testa a trava novamente.

Tendo em vista que ***thread_yield* é apenas uma chamada para o escalonador de *threads* no espaço de usuário, ela é muito rápida**. Em consequência, **nem *mutex_lock*, tampouco *mutex_unlock* exigem quaisquer chamadas de núcleo**. **Usando-os, *threads* de usuário podem sincronizar inteiramente no espaço do usuário usando rotinas.**

2.3.7 Monitores

Brinch Hansen (1973) e Hoare (1974) propuseram uma primitiva de sincronização de nível mais alto chamada monitor. Um monitor é uma coleção de rotinas, variáveis e estruturas de dados que são reunidas em um tipo especial de módulo ou pacote. Processos podem chamar as rotinas em um monitor sempre que eles quiserem, mas eles não podem acessar diretamente as estruturas de dados internos do monitor a partir de rotinas declaradas fora dele. A Figura 2.33 ilustra um monitor.

Os monitores têm uma propriedade importante que os torna úteis para realizar a exclusão mútua: apenas um processo pode estar ativo em um monitor em qualquer dado instante. Quando um processo chama uma rotina do monitor, as primeiras instruções conferirão para ver se qualquer outro processo está ativo no momento dentro do monitor. Se isso ocorrer, o processo que chamou será suspenso até que o outro processo tenha deixado o monitor. Se nenhum outro processo está usando o monitor, o processo que chamou pode entrar.

Cabe ao compilador implementar a exclusão mútua nas entradas do monitor, mas uma maneira comum é usar um *mutex* ou um semáforo binário. Como o compilador, não o programador, está arranjando a exclusão mútua, é muito menos provável que algo dê errado. De qualquer

maneira, a pessoa escrevendo o monitor não precisa ter ciência de como o compilador arranja a exclusão mútua. Basta saber que ao transformar todas as regiões críticas em rotinas de monitores, dois processos jamais executarão suas regiões críticas ao mesmo tempo.

Embora os monitores proporcionem uma maneira de alcançar a exclusão mútua, isso não é suficiente. É necessário que os processos sejam bloqueados quando não puderem prosseguir. No problema do produtor-consumidor, pode-se colocar todos os testes de *buffer* cheio e *buffer* vazio nas rotinas de monitor. Mas como o produtor seria bloqueado quando encontrasse o *buffer* cheio?

A solução encontra-se na introdução de **variáveis de condição**, junto com duas operações, ***wait* e *signal***. Quando uma rotina de monitor descobre que não pode continuar (por exemplo, o produtor encontra o *buffer* cheio), ela realiza um *wait* em alguma variável de condição, digamos, *full*. Essa ação provoca o bloqueio do processo que está chamando. Ele também permite outro processo que tenha sido previamente proibido de entrar no monitor a entrar agora.

FIGURA 2.33 Um monitor.

```
monitor example
    integer i;
    condition c;

    procedure producer ( );
    .
    .
    .
    end;

    procedure consumer ( );
    .       .       .

    end;
end monitor;
```

Nesse outro processo, o consumidor, por exemplo, pode despertar o parceiro adormecido realizando um *signal* na variável de condição que seu parceiro está esperando. Para evitar ter dois processos ativos no monitor ao mesmo tempo, precisamos de uma regra dizendo o que acontece depois de um *signal*. Brinch **Hansen propôs uma saída inteligente para o problema, exigindo que um processo realizando um *signal* deva sair do monitor imediatamente.** Em outras palavras, um comando *signal* pode aparecer apenas como o comando final em uma rotina de monitor. Se um *signal* for realizado em uma variável de condição em que vários processos estejam esperando, apenas um deles, determinado pelo escalonador do sistema, será revivido. Um esqueleto do problema produtor-consumidor com monitores é dado na Figura 2.34.

FIGURA 2.34 Um esqueleto do problema produtor-consumidor com monitores. Somente uma rotina do monitor está ativa por vez. O buffer tem *N* vagas.

```
monitor ProducerConsumer
    condition full, empty;
    integer count,

    procedure insert(item: integer);
    begin
        if count= N then wait (full);
        insert_item(item);
        count: = count + 1;
        if count = 1then signal(empty)
    end ;

    function remove:integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count: = count– 1;
        if count  = N – 1 then signal(full)
    end;

    count:= 0;
end monitor;

procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end ;
```

Talvez você esteja pensando que as operações *wait* e *signal* parecem similares a *sleep* e *wakeup*, que vimos antes e tinham condições de corrida fatais. Bem, elas são muito similares, mas com uma diferença crucial: *sleep* e *wakeup* fracassaram porque enquanto um processo estava tentando dormir, o outro tentava despertá-lo. Com monitores, isso não pode acontecer. A exclusão mútua automática nas rotinas de monitor garante que se, digamos, o produtor dentro de uma rotina de monitor descobrir que o *buffer* está cheio, ele será capaz de completar a operação *wait* sem ter de se preocupar com a possibilidade de que o escalonador possa trocar para o consumidor um instante antes de *wait* ser concluída. O consumidor não será nem deixado entrar no monitor até que *wait* seja concluído e o produtor seja marcado como não mais executável.

Ao tornar automática a exclusão mútua de regiões críticas, os monitores tornam a programação paralela muito menos propensa a erros do que o uso de semáforos. Tanto monitores, quanto semáforos, foram projetados para solucionar o problema da exclusão mútua em uma ou mais CPUs, todas com acesso a uma memória comum.

2.3.8 Barreiras

Este mecanismo de sincronização é dirigido a grupos de processos em vez de situações que envolvem dois processos do tipo produtor-consumidor. Algumas aplicações são divididas em fases e têm como regra que nenhum processo deve prosseguir para a fase seguinte até que todos os processos estejam prontos para isso. Tal comportamento pode ser conseguido colocando uma barreira no fim de cada fase. Quando um processo atinge a barreira, ele é bloqueado até que todos os processos tenham atingido a barreira. Isso permite que grupos de processos sincronizem. A operação de barreira está ilustrada na Figura 2.37.

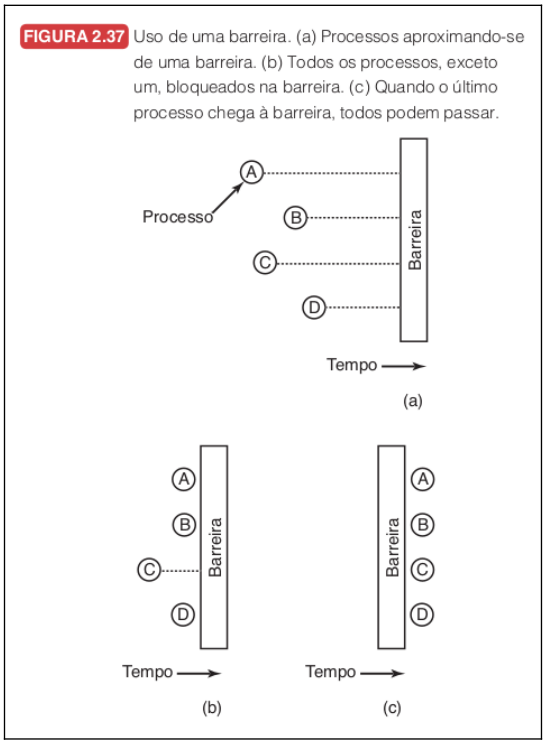


FIGURA 2.37 Uso de uma barreira. (a) Processos aproximando-se de uma barreira. (b) Todos os processos, exceto um, bloqueados na barreira. (c) Quando o último processo chega à barreira, todos podem passar.

Na Figura 2.37(a) vemos quatro processos aproximando-se de uma barreira. O que isso significa é que eles estão apenas computando e não chegaram ao fim da fase atual ainda. Após um tempo, o primeiro processo termina toda a computação exigida dele durante a primeira fase, então, ele executa a primitiva *barrier*, geralmente chamando um procedimento de biblioteca. O processo é então suspenso. Um pouco mais tarde, um segundo e então um terceiro processo terminam a primeira fase e também executam a primitiva *barrier*. Essa situação está ilustrada na Figura 2.37(b). Por fim, quando o último processo C, atinge a barreira, todos os processos são liberados, como mostrado na Figura 2.37(c). Como exemplo de um problema exigindo barreiras, considere um problema típico de relaxação na física ou engenharia.

2.4 Escalonamento

2.4.1 Introdução ao escalonamento

2.4.2 Escalonamento em sistemas em lote

2.4.3 Escalonamento em sistemas interativos

2.4.4 Escalonamento em sistemas de tempo real

2.4.5 Política versus mecanismo

2.4.6 Escalonamento de threads