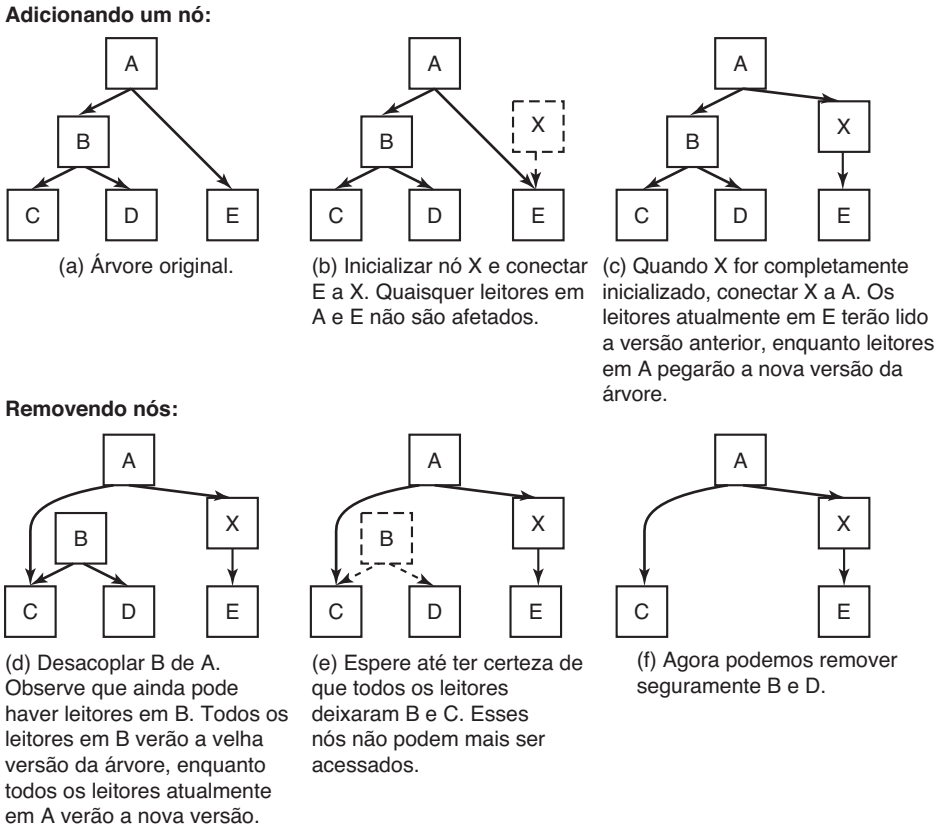
FIGURA 2.38 *Leitura-cópia-atualização*: inserindo um nó na árvore e então removendo um galho — tudo sem travas.



tenha deixado esses nós. RCU determina cuidadosamente o tempo máximo que um leitor pode armazenar uma referência para a estrutura de dados. Após esse período, ele pode recuperar seguramente a memória. Especificamente, leitores acessam a estrutura de dados no que é conhecido como uma **seção crítica do lado do leitor** que pode conter qualquer código, desde que não bloqueie ou adormeça. Nesse caso, sabemos o tempo máximo que precisamos esperar. Especificamente, definimos um **período de graça** como qualquer período no qual sabemos que cada thread está do lado de fora da seção de leitura crítica pelo menos uma vez. Tudo ficará bem se esperarmos por um período que seja pelo menos igual ao período de graça antes da recuperação. Como não é permitido ao código na seção crítica de leitura bloquear ou adormecer, um critério simples é esperar até que todos os threads tenham realizado um chaveamento de contexto.

2.4 Escalonamento

Quando um computador é multiprogramado, ele frequentemente tem múltiplos processos ou threads competindo pela CPU ao mesmo tempo. Essa situação ocorre sempre que dois ou mais deles estão simultaneamente no estado pronto. Se apenas uma CPU está disponível, uma escolha precisa ser feita sobre qual processo será executado em seguida. A parte do sistema operacional que faz a escolha é chamada de **escalonador**, e o algoritmo que ele usa é chamado de **algoritmo de escalonamento**. Esses tópicos formam o assunto a ser tratado nas seções a seguir.

Muitas das mesmas questões que se aplicam ao escalonamento de processos também se aplicam ao escalonamento de threads, embora algumas sejam diferentes. Quando o núcleo gerencia threads, o escalonamento é geralmente feito por thread, com pouca ou nenhuma consideração sobre o processo ao qual o thread pertence. De início nos concentraremos nas questões de escalonamento que se aplicam a ambos, processos e threads. Depois, examinaremos explicitamente o escalonamento de threads e algumas das questões exclusivas que ele gera. Abordaremos os chips multinúcleo no Capítulo 8.

2.4.1 Introdução ao escalonamento

Nos velhos tempos dos sistemas em lote com a entrada na forma de imagens de cartões em uma fita

magnética, o algoritmo de escalonamento era simples: apenas execute o próximo trabalho na fita. Com os sistemas de multiprogramação, ele tornou-se mais complexo porque geralmente havia múltiplos usuários esperando pelo serviço. Alguns computadores de grande porte ainda combinam serviço em lote e de compartilhamento de tempo, exigindo que o escalonador decida se um trabalho em lote ou um usuário interativo em um terminal deve ir em seguida. (Como uma nota, um trabalho em lote pode ser uma solicitação para executar múltiplos programas em sucessão, mas para esta seção presumiremos apenas que se trata de uma solicitação para executar um único programa.) Como o tempo de CPU é um recurso escasso nessas máquinas, um bom escalonador pode fazer uma grande diferença no desempenho percebido e satisfação do usuário. Em consequência, uma grande quantidade de trabalho foi dedicada ao desenvolvimento de algoritmos de escalonamento inteligentes e eficientes.

Com o advento dos computadores pessoais, a situação mudou de duas maneiras. Primeiro, na maior parte do tempo há apenas um processo ativo. É improvável que um usuário preparando um documento em um processador de texto esteja simultaneamente compilando um programa em segundo plano. Quando o usuário digita um comando ao processador de texto, o escalonador não precisa fazer muito esforço para descobrir qual processo executar — o processador de texto é o único candidato.

Segundo, computadores tornaram-se tão rápidos com o passar dos anos que a CPU dificilmente ainda é um recurso escasso. A maioria dos programas para computadores pessoais é limitada pela taxa na qual o usuário pode apresentar a entrada (digitando ou clicando), não pela taxa na qual a CPU pode processá-la. Mesmo as compilações, um importante sorvedouro de ciclos de CPUs no passado, levam apenas alguns segundos hoje. Mesmo quando dois programas estão de fato sendo executados ao mesmo tempo, como um processador de texto e uma planilha, dificilmente importa qual deles vai primeiro, pois o usuário provavelmente está esperando que ambos terminem. Como consequência, o escalonamento não importa muito em PCs simples. É claro que há aplicações que praticamente devoram a CPU viva. Por exemplo, reproduzir uma hora de vídeo de alta resolução enquanto se ajustam as cores em cada um dos 107.892 quadros (em NTSC) ou 90.000 quadros (em PAL) exige uma potência computacional de nível industrial. No entanto, aplicações similares são a exceção em vez de a regra.

Quando voltamos aos servidores em rede, a situação muda consideravelmente. Aqui múltiplos processos muitas vezes competem pela CPU, de maneira que o escalonamento importa outra vez. Por exemplo, quando a CPU tem de escolher entre executar um processo que reúne as estatísticas diárias e um que serve a solicitações de usuários, estes ficarão muito mais contentes se o segundo receber a primeira chance de acessar a CPU.

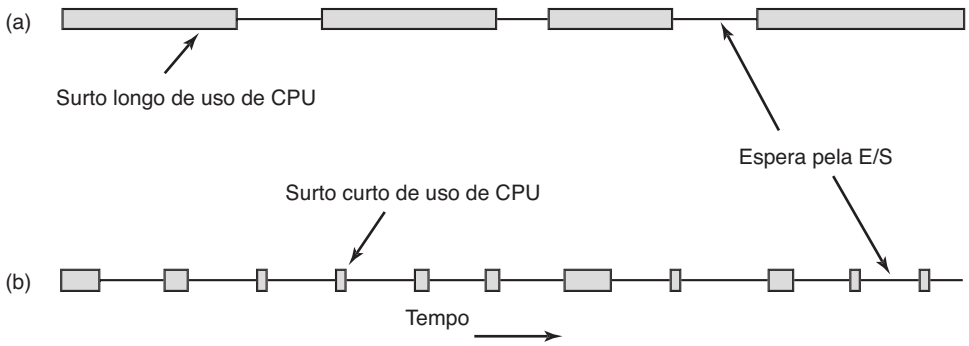
O argumento da "abundância de recursos" também não se sustenta em muitos dispositivos móveis, como smartphones (exceto talvez os modelos mais potentes) e nós em redes de sensores. Além disso, já que a duração da bateria é uma das restrições mais importantes nesses dispositivos, alguns escalonadores tentam otimizar o consumo de energia.

Além de escolher o processo certo a ser executado, o escalonador também tem de se preocupar em fazer um uso eficiente da CPU, pois o chaveamento de processos é algo caro. Para começo de conversa, uma troca do modo usuário para o modo núcleo precisa ocorrer. Então o estado do processo atual precisa ser salvo, incluindo armazenar os seus registros na tabela de processos para que eles possam ser recarregados mais tarde. Em alguns sistemas, o mapa de memória (por exemplo, os bits de referência à memória na tabela de páginas) precisa ser salvo da mesma maneira. Em seguida, um novo processo precisa ser selecionado executando o algoritmo de escalonamento. Após isso, a MMU (memory management unit — unidade de gerenciamento de memória) precisa ser recarregada com o mapa de memória do novo processo. Por fim, o novo processo precisa ser inicializado. Além de tudo isso, a troca de processo pode invalidar o cache de memória e as tabelas relacionadas, forçando-o a ser dinamicamente recarregado da memória principal duas vezes (ao entrar no núcleo e ao deixá-lo). De modo geral, realizar muitas trocas de processos por segundo pode consumir um montante substancial do tempo da CPU, então, recomenda-se cautela.

Comportamento de processos

Quase todos os processos alternam surtos de computação com solicitações de E/S (disco ou rede), como mostrado na Figura 2.39. Muitas vezes, a CPU executa por um tempo sem parar, então uma chamada de sistema é feita para ler de um arquivo ou escrever para um arquivo. Quando a chamada de sistema é concluída, a CPU calcula novamente até que ela precisa de mais dados ou tem de escrever mais dados, e assim por diante. Observe que algumas atividades de E/S contam como computação. Por exemplo, quando a CPU copia bits para uma RAM de vídeo para atualizar a tela, ela está

FIGURA 2.39 Surtos de uso da CPU alternam-se com períodos de espera por E/S. (a) Um processo limitado pela CPU. (b) Um processo limitado pela E/S.



computando, não realizando E/S, pois a CPU está em uso. E/S nesse sentido é quando um processo entra no estado bloqueado esperando por um dispositivo externo para concluir o seu trabalho.

A questão importante a ser observada a respeito da Figura 2.39 é que alguns processos, como o mostrado na Figura 2.39(a), passam a maior do tempo computando, enquanto outros, como o mostrado na Figura 2.39(b), passam a maior parte do tempo esperando pela E/S. Os primeiros são chamados **limitados pela computação** ou **limitados pela CPU**; os segundos são chamados **limitados pela E/S**. Processos limitados pela CPU geralmente têm longos surtos de CPU e então esporádicas esperas de E/S, enquanto os processos limitados pela E/S têm surtos de CPU curtos e esperas de E/S frequentes. Observe que o fator chave é o comprimento do surto da CPU, não o comprimento do surto da E/S. Processos limitados pela E/S são limitados pela E/S porque eles não computam muito entre solicitações de E/S, não por terem tais solicitações especialmente demoradas. Eles levam o mesmo tempo para emitir o pedido de hardware para ler um bloco de disco, independentemente de quanto tempo levam para processar os dados após eles chegarem.

Vale a pena observar que, à medida que as CPUs ficam mais rápidas, os processos tendem a ficar mais limitados pela E/S. Esse efeito ocorre porque as CPUs estão melhorando muito mais rápido que os discos. Em consequência, é provável que o escalonamento de processos limitados pela E/S torne-se um assunto mais importante no futuro. A ideia básica aqui é que se um processo limitado pela E/S quiser executar, ele deve receber uma chance rapidamente para que possa emitir sua solicitação de disco e manter o disco ocupado. Como vimos na Figura 2.6, quando os processos são limitados pela E/S, são necessários diversos deles para manter a CPU completamente ocupada.

Quando escalonar

Uma questão fundamental relacionada com o escalonamento é quando tomar decisões de escalonamento. Na realidade, há uma série de situações nas quais o escalonamento é necessário. Primeiro, quando um novo processo é criado, uma decisão precisa ser tomada a respeito de qual processo, o pai ou o filho, deve ser executado. Tendo em vista que ambos os processos estão em um estado pronto, trata-se de uma decisão de escalonamento normal e pode ser qualquer uma, isto é, o escalonador pode legitimamente escolher executar o processo pai ou o filho em seguida.

Segundo, uma decisão de escalonamento precisa ser tomada ao término de um processo. Esse processo não pode mais executar (já que ele não existe mais), então algum outro precisa ser escolhido do conjunto de processos prontos. Se nenhum está pronto, um processo ocioso gerado pelo sistema normalmente é executado.

Terceiro, quando um processo bloqueia para E/S, em um semáforo, ou por alguma outra razão, outro processo precisa ser selecionado para executar. Às vezes, a razão para bloquear pode ter um papel na escolha. Por exemplo, se *A* é um processo importante e ele está esperando por *B* para sair de sua região crítica, deixar que *B* execute em seguida permitirá que ele saia de sua região crítica e desse modo deixe que *A* continue. O problema, no entanto, é que o escalonador geralmente não tem a informação necessária para levar essa dependência em consideração.

Quarto, quando ocorre uma interrupção de E/S, uma decisão de escalonamento pode ser feita. Se a interrupção veio de um dispositivo de E/S que agora completou seu trabalho, algum processo que foi bloqueado esperando pela E/S pode agora estar pronto para executar. Cabe ao escalonador decidir se deve executar o processo que ficou pronto há pouco, o processo que estava

sendo executado no momento da interrupção, ou algum terceiro processo.

Se um hardware de relógio fornece interrupções periódicas a 50 ou 60 Hz ou alguma outra frequência, uma decisão de escalonamento pode ser feita a cada interrupção ou a cada *k*-ésima interrupção de relógio. Algoritmos de escalonamento podem ser divididos em duas categorias em relação a como lidar com interrupções de relógio. Um algoritmo de escalonamento **não preemptivo** escolhe um processo para ser executado e então o deixa ser executado até que ele seja bloqueado (seja em E/S ou esperando por outro processo), ou libera voluntariamente a CPU. Mesmo que ele execute por muitas horas, não será suspenso forçosamente. Na realidade, nenhuma decisão de escalonamento é feita durante interrupções de relógio. Após o processamento da interrupção de relógio ter sido concluído, o processo que estava executando antes da interrupção é retomado, a não ser que um processo mais prioritário esteja esperando por um tempo de espera agora satisfeito.

Por outro lado, um algoritmo de escalonamento **preemptivo** escolhe um processo e o deixa executar por no máximo um certo tempo fixado. Se ele ainda estiver executando ao fim do intervalo de tempo, ele é suspenso e o escalonador escolhe outro processo para executar (se algum estiver disponível). Realizar o escalonamento preemptivo exige que uma interrupção de relógio ocorra ao fim do intervalo para devolver o controle da CPU de volta para o escalonador. Se nenhum relógio estiver disponível, o escalonamento não preemptivo é a única solução.

Categorias de algoritmos de escalonamento

De maneira pouco surpreendente, em diferentes ambientes, distintos algoritmos de escalonamento são necessários. Essa situação surge porque diferentes áreas de aplicação (e de tipos de sistemas operacionais) têm metas diversas. Em outras palavras, o que deve ser otimizado pelo escalonador não é o mesmo em todos os sistemas. Três ambientes valem ser destacados aqui:

1. Lote.
2. Interativo.
3. Tempo real.

Sistemas em lote ainda são amplamente usados no mundo de negócios para folhas de pagamento, estoques, contas a receber, contas a pagar, cálculos de juros (em bancos), processamento de pedidos de indenização (em companhias de seguro) e outras tarefas periódicas. Em sistemas em lote, não há usuários esperando impacientemente em seus terminais para uma resposta rápida a uma solicitação menor. Em consequência, algoritmos não preemptivos, ou algoritmos preemptivos com longos períodos para cada processo são muitas vezes aceitáveis. Essa abordagem reduz os chaveamentos de processos e melhora o desempenho. Na realidade, os algoritmos em lote são bastante comuns e muitas vezes aplicáveis a outras situações também, o que torna seu estudo interessante, mesmo para pessoas não envolvidas na computação corporativa de grande porte.

Em um ambiente com usuários interativos, a preempção é essencial para evitar que um processo tome conta da CPU e negue serviço para os outros. Mesmo que nenhum processo execute de modo intencional para sempre, um erro em um programa pode levar um processo a impedir indefinidamente que todos os outros executem. A preempção é necessária para evitar esse comportamento. Os servidores também caem nessa categoria, visto que eles normalmente servem a múltiplos usuários (remotos), todos os quais estão muito apressados, assim como usuários de computadores.

Em sistemas com restrições de tempo real, a preempção às vezes, por incrível que pareça, não é necessária, porque os processos sabem que eles não podem executar por longos períodos e em geral realizam o seu trabalho e bloqueiam rapidamente. A diferença com os sistemas interativos é que os de tempo real executam apenas programas que visam ao progresso da aplicação à mão. Sistemas interativos são sistemas para fins gerais e podem executar programas arbitrários que não são cooperativos e talvez até mesmo maliciosos.

Objetivos do algoritmo de escalonamento

A fim de projetar um algoritmo de escalonamento, é necessário ter alguma ideia do que um bom algoritmo deve fazer. Certas metas dependem do ambiente (em lote, interativo ou de tempo real), mas algumas são desejáveis em todos os casos. Algumas metas estão listadas na Figura 2.40. Discutiremos essas metas a seguir.

Em qualquer circunstância, a justiça é importante. Processos comparáveis devem receber serviços comparáveis. Conceder a um processo muito mais tempo de CPU do que para um processo equivalente não é justo. É claro que categorias diferentes de processos podem ser tratadas diferentemente. Pense sobre controle de segurança e elaboração da folha de pagamento em um centro de computadores de um reator nuclear.

De certa maneira relacionado com justiça está o cumprimento das políticas do sistema. Se a política local é que os processos de controle de segurança são executados

FIGURA 2.40 Algumas metas do algoritmo de escalonamento sob diferentes circunstâncias.

Todos os sistemas
Justiça — dar a cada processo uma porção justa da CPU
Aplicação da política — verificar se a política estabelecida é cumprida
Equilíbrio — manter ocupadas todas as partes do sistema
Sistemas em lote
Vazão (*throughput*) — maximizar o número de tarefas por hora
Tempo de retorno — minimizar o tempo entre a submissão e o término
Utilização de CPU — manter a CPU ocupada o tempo todo
Sistemas interativos
Tempo de resposta — responder rapidamente às requisições
Proporcionalidade — satisfazer às expectativas dos usuários
Sistemas de tempo real
Cumprimento dos prazos — evitar a perda de dados
Previsibilidade — evitar a degradação da qualidade em sistemas multimídia

sempre que quiserem, mesmo que isso signifique atraso de 30 segundos da folha de pagamento, o escalonador precisa certificar-se de que essa política seja cumprida.

Outra meta geral é manter todas as partes do sistema ocupadas quando possível. Se a CPU e todos os outros dispositivos de E/S podem ser mantidos executando o tempo inteiro, mais trabalho é realizado por segundo do que se alguns dos componentes estivessem ociosos. No sistema em lote, por exemplo, o escalonador tem controle sobre quais tarefas são trazidas à memória para serem executadas. Ter alguns processos limitados pela CPU e alguns limitados pela E/S juntos na memória é uma ideia melhor do que primeiro carregar e executar todas as tarefas limitadas pela CPU e, quando forem concluídas, carregar e executar todas as tarefas limitadas pela E/S. Se a segunda estratégia for usada, quando os processos limitados pela CPU estiverem sendo executados, eles disputarão a CPU e o disco ficará ocioso. Depois, quando as tarefas limitadas pela E/S entrarem, elas disputarão o disco e a CPU ficará ociosa. Assim, é melhor manter o sistema inteiro executando ao mesmo tempo mediante uma mistura cuidadosa de processos.

Os gerentes de grandes centros de computadores que executam muitas tarefas em lote costumam observar três métricas para ver como seus sistemas estão desempenhando: vazão, tempo de retorno e utilização da CPU. A **vazão** é o número de tarefas por hora que o sistema completa. Considerados todos os fatores, terminar 50 tarefas por hora é melhor do que terminar 40 tarefas por hora. O **tempo de retorno** é estatisticamente o tempo médio do momento em que a tarefa em lote é submetida até o momento em que ela é concluída. Ele mede quanto tempo o usuário médio tem de esperar pela saída. Aqui a regra é: menos é mais.

Um algoritmo de escalonamento que tenta maximizar a vazão talvez não minimize necessariamente o tempo de retorno. Por exemplo, dada uma combinação de tarefas curtas e tarefas longas, um escalonador que sempre executou tarefas curtas e nunca as longas talvez consiga uma excelente vazão (muitas tarefas curtas por hora), mas à custa de um tempo de retorno terrível para as tarefas longas. Se as tarefas curtas seguissem chegando a uma taxa aproximadamente uniforme, as tarefas longas talvez nunca fossem executadas, tornando o tempo de retorno médio infinito, conquanto alcançando uma alta vazão.

A utilização da CPU é muitas vezes usada como uma métrica nos sistemas em lote. No entanto, ela não é uma boa métrica. O que de fato importa é quantas tarefas por hora saem do sistema (vazão) e quanto tempo leva para receber uma tarefa de volta (tempo de retorno). Usar a utilização de CPU como uma métrica é como classificar carros com base em seu giro de motor. Entretanto, saber quando a utilização da CPU está próxima de 100% é útil para saber quando chegou o momento de obter mais poder computacional.

Para sistemas interativos, aplicam-se metas diferentes. A mais importante é minimizar o **tempo de resposta**, isto é, o tempo entre emitir um comando e receber o resultado. Em um computador pessoal, em que um processo de segundo plano está sendo executado (por exemplo, lendo e armazenando e-mail da rede), uma solicitação de usuário para começar um programa ou abrir um arquivo deve ter precedência sobre o trabalho de segundo plano. Atender primeiro todas as solicitações interativas será percebido como um bom serviço.

Uma questão de certa maneira relacionada é o que poderia ser chamada de **proporcionalidade**. Usuários têm uma ideia inerente (porém muitas vezes incorreta) de quanto tempo as coisas devem levar. Quando uma solicitação que o usuário percebe como complexa leva muito tempo, os usuários aceitam isso, mas quando uma solicitação percebida como simples leva muito tempo, eles ficam irritados. Por exemplo, se clicar em um ícone que envia um vídeo de 500 MB para um servidor na nuvem demorar 60 segundos, o usuário provavelmente aceitará isso como um fato da vida por não esperar que a transferência leve 5 s. Ele sabe que levará um tempo.

Por outro lado, quando um usuário clica em um ícone que desconecta a conexão com o servidor na nuvem após o vídeo ter sido enviado, ele tem expectativas diferentes. Se a desconexão não estiver completa após 30 s, o usuário provavelmente estará soltando algum palavrão e após 60 s ele estará espumando de raiva. Esse comportamento decorre da percepção comum dos usuários de que enviar um monte de dados *supostamente* leva muito mais tempo que apenas desconectar uma conexão. Em alguns casos (como esse), o escalonador não pode fazer nada a respeito do tempo de resposta, mas em outros casos ele pode, especialmente quando o atraso é causado por uma escolha ruim da ordem dos processos.

Sistemas de tempo real têm propriedades diferentes de sistemas interativos e, desse modo, metas de escalonamento diferentes. Eles são caracterizados por ter prazos que devem — ou pelo menos deveriam —, ser cumpridos. Por exemplo, se um computador está controlando um dispositivo que produz dados a uma taxa regular, deixar de executar o processo de coleta de dados em tempo pode resultar em dados perdidos. Assim, a principal exigência de um sistema de tempo real é cumprir com todos (ou a maioria) dos prazos.

Em alguns sistemas de tempo real, especialmente aqueles envolvendo multimídia, a previsibilidade é importante. Descumprir um prazo ocasional não é fatal, mas se o processo de áudio executar de maneira errática demais, a qualidade do som deteriorará rapidamente. O vídeo também é uma questão, mas o ouvido é muito mais sensível a atrasos que o olho. Para evitar esse problema, o escalonamento de processos deve ser altamente previsível e regular. Neste capítulo, estudaremos algoritmos de escalonamento interativo e em lote. O escalonamento de tempo real não é abordado no livro, mas no material extra sobre sistemas operacionais de multimídia na Sala Virtual do livro.

2.4.2 Escalonamento em sistemas em lote

Chegou o momento agora de passar das questões de escalonamento gerais para algoritmos de escalonamento específicos. Nesta seção, examinaremos os algoritmos usados em sistemas em lote. Nas seções seguintes, examinaremos sistemas interativos e de tempo real. Vale a pena destacar que alguns algoritmos são usados tanto nos sistemas interativos como nos em lote. Estudaremos esses mais tarde.

Primeiro a chegar, primeiro a ser servido

É provável que o mais simples de todos os algoritmos de escalonamento já projetados seja o **primeiro a chegar, primeiro a ser servido** (**first-come, first-served**) não preemptivo. Com esse algoritmo, a CPU é atribuída aos processos na ordem em que a requisitam. Basicamente, há uma fila única de processos prontos. Quando a primeira tarefa entrar no sistema de manhã, ela é iniciada imediatamente e deixada executar por quanto tempo ela quiser. Ela não é interrompida por ter sido executada por tempo demais. À medida que as outras tarefas chegam, elas são colocadas no fim da fila. Quando o processo que está sendo executado é bloqueado, o primeiro processo na fila é executado em seguida. Quando um processo bloqueado fica pronto — assim como uma tarefa que chegou há pouco —, ele é colocado no fim da fila, atrás dos processos em espera.

A grande força desse algoritmo é que ele é fácil de compreender e igualmente fácil de programar. Ele também é tão justo quanto alocar ingressos escassos de um concerto ou iPhones novos para pessoas que estão dispostas a esperar na fila desde às duas da manhã. Com esse algoritmo, uma única lista encadeada controla todos os processos. Escolher um processo para executar exige apenas remover um da frente da fila. Acrescentar uma nova tarefa ou desbloquear um processo exige apenas colocá-lo no fim da fila. O que poderia ser mais simples de compreender e implementar?

Infelizmente, o primeiro a chegar, primeiro a ser servido também tem uma desvantagem poderosa. Suponha que há um processo limitado pela computação que é executado por 1 s de cada vez e muitos processos limitados pela E/S que usam pouco tempo da CPU, mas cada um tem de realizar 1.000 leituras do disco para ser concluído. O processo limitado pela computação é executado por 1 s, então ele lê um bloco de disco. Todos os processos de E/S são executados agora e começam leituras de disco. Quando o processo limitado pela computação obtém seu bloco de disco, ele é executado por mais 1 s, seguido por todos os processos limitados pela E/S em rápida sucessão.

O resultado líquido é que cada processo limitado pela E/S lê 1 bloco por segundo e levará 1.000 s para terminar. Com o algoritmo de escalonamento que causasse a preempção do processo limitado pela computação a cada 10 ms, os processos limitados pela E/S terminariam em 10 s em vez de 1.000 s, e sem retardar muito o processo limitado pela computação.

Tarefa mais curta primeiro

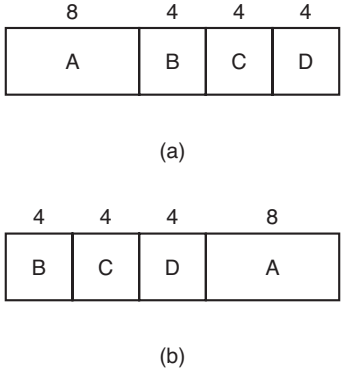
Agora vamos examinar outro algoritmo em lote não preemptivo que presume que os tempos de execução são conhecidos antecipadamente. Em uma companhia de seguros, por exemplo, as pessoas podem prever com

bastante precisão quanto tempo levará para executar um lote de 1.000 solicitações, tendo em vista que um trabalho similar é realizado todos os dias. Quando há vários trabalhos igualmente importantes esperando na fila de entrada para serem iniciados, o escalonador escolhe a **tarefa mais curta primeiro** (**shortest job first**). Observe a Figura 2.41. Nela vemos quatro tarefas *A, B, C* e *D* com tempos de execução de 8, 4, 4 e 4 minutos, respectivamente. Ao executá-las nessa ordem, o tempo de retorno para *A* é 8 minutos, para *B* é 12 minutos, para *C* é 16 minutos e para *D* é 20 minutos, resultando em uma média de 14 minutos.

Agora vamos considerar executar essas quatro tarefas usando o algoritmo *tarefa mais curta primeiro*, como mostrado na Figura 2.41(b). Os tempos de retorno são agora 4, 8, 12 e 20 minutos, resultando em uma média de 11 minutos. A tarefa mais curta primeiro é provavelmente uma ótima escolha. Considere o caso de quatro tarefas, com tempos de execução de *a, b, c* e *d*, respectivamente. A primeira tarefa termina no tempo *a*, a segunda no tempo $a + b$, e assim por diante. O tempo de retorno médio é $(4a + 3b + 2c + d)/4$. Fica claro que *a* contribui mais para a média do que os outros tempos, logo deve ser a tarefa mais curta, com *b* em seguida, então *c* e finalmente *d* como o mais longo, visto que ele afeta apenas seu próprio tempo de retorno. O mesmo argumento aplica-se igualmente bem a qualquer número de tarefas.

Vale a pena destacar que a *tarefa mais curta primeiro* é ótima apenas quando todas as tarefas estão disponíveis simultaneamente. Como um contraexemplo, considere cinco tarefas, *A* a *E*, com tempos de execução de 2, 4, 1, 1 e 1, respectivamente. Seus tempos de chegada são 0, 0, 3, 3 e 3. No início, apenas *A* ou *B* podem ser escolhidos, dado que as outras três tarefas não chegaram ainda. Usando a *tarefa mais curta primeiro*, executaremos as tarefas na ordem *A, B, C, D, E* para um tempo de espera médio de 4,6. No entanto, executá-las na ordem *B, C, D, E, A* tem um tempo de espera médio de 4,4.

FIGURA 2.41 Um exemplo do escalonamento tarefa mais curta primeiro. (a) Executando quatro tarefas na ordem original. (b) Executando-as na ordem tarefa mais curta primeiro.



Tempo restante mais curto em seguida

Uma versão preemptiva da ***tarefa mais curta primeiro*** é o **tempo restante mais curto em seguida** (*shortest remaining time next)*. Com esse algoritmo, o escalonador escolhe o processo cujo tempo de execução restante é o mais curto. De novo, o tempo de execução precisa ser conhecido antecipadamente. Quando uma nova tarefa chega, seu tempo total é comparado com o tempo restante do processo atual. Se a nova tarefa precisa de menos tempo para terminar do que o processo atual, este é suspenso e a nova tarefa iniciada. Esse esquema permite que tarefas curtas novas tenham um bom desempenho.

2.4.3 Escalonamento em sistemas interativos

Examinaremos agora alguns algoritmos que podem ser usados em sistemas interativos. Eles são comuns em computadores pessoais, servidores e outros tipos de sistemas também.

Escalonamento por chaveamento circular

Um dos algoritmos mais antigos, simples, justos e amplamente usados é o **circular** (*round-robin*). A cada processo é designado um intervalo, chamado de seu **quantum**, durante o qual ele é deixado executar. Se o processo ainda está executando ao fim do quantum, a CPU sofrerá uma preempção e receberá outro processo. Se o processo foi bloqueado ou terminado antes de o quantum ter decorrido, o chaveamento de CPU será feito quando o processo bloquear, é claro. O escalonamento circular é fácil de implementar. Tudo o que o escalonador precisa fazer é manter uma lista de processos executáveis, como mostrado na Figura 2.42(a). Quando o processo usa todo o seu quantum, ele é colocado no fim da lista, como mostrado na Figura 2.42(b).

A única questão realmente interessante em relação ao escalonamento circular é o comprimento do quantum. Chavear de um processo para o outro exige certo

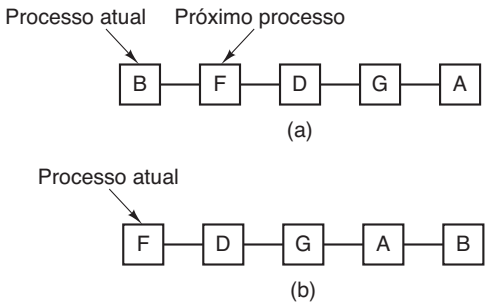


FIGURA 2.42 Escalonamento circular. (a) A lista de processos executáveis. (b) A lista de processos executáveis após *B* usar todo seu quantum.

montante de tempo para fazer toda a administração — salvando e carregando registradores e mapas de memória, atualizando várias tabelas e listas, carregando e descarregando memória cache, e assim por diante. Suponha que esse **chaveamento de processo** ou **chaveamento de contexto**, como é chamado às vezes, leva 1 ms, incluindo o chaveamento dos mapas de memória, carregar e descarregar o cache etc. Também suponha que o quantum é estabelecido em 4 ms. Com esses parâmetros, após realizar 4 ms de trabalho útil, a CPU terá de gastar (isto é, desperdiçar) 1 ms no chaveamento de processo. Desse modo, 20% do tempo da CPU será jogado fora em overhead administrativo. Claramente, isso é demais.

Para melhorar a eficiência da CPU, poderíamos configurar o quantum para, digamos, 100 ms. Agora o tempo desperdiçado é de apenas 1%. Mas considere o que acontece em um sistema de servidores se 50 solicitações entram em um intervalo muito curto e com exigências de CPU com grande variação. Cinquenta processos serão colocados na lista de processos executáveis. Se a CPU estiver ociosa, o primeiro começará imediatamente, o segundo não poderá começar até 100 ms mais tarde e assim por diante. O último azarado talvez tenha de esperar 5 s antes de ter uma chance, presumindo que todos os outros usem todo o seu quantum. A maioria dos usuários achará demorada uma resposta de 5 s para um comando curto. Essa situação seria especialmente ruim se algumas das solicitações próximas do fim da fila exigissem apenas alguns milissegundos de tempo da CPU. Com um quantum curto, eles teriam recebido um serviço melhor.

Outro fator é que se o quantum for configurado por um tempo mais longo que o surto de CPU médio, a preempção não acontecerá com muita frequência. Em vez disso, a maioria dos processos desempenhará uma operação de bloqueio antes de o quantum acabar, provocando um chaveamento de processo. Eliminar a preempção melhora o desempenho, porque os chaveamentos de processo então acontecem apenas quando são logicamente necessários, isto é, quando um processo é bloqueado e não pode continuar.

A seguinte conclusão pode ser formulada: estabelecer o quantum curto demais provoca muitos chaveamentos de processos e reduz a eficiência da CPU, mas estabelecê-lo longo demais pode provocar uma resposta ruim a solicitações interativas curtas. Um quantum em torno de 20-50 ms é muitas vezes bastante razoável.

Escalonamento por prioridades

O escalonamento circular pressupõe implicitamente que todos os processos são de igual importância. Muitas vezes, as pessoas que são proprietárias e operam computadores multiusuário têm ideias bem diferentes sobre o assunto. Em uma universidade, por exemplo, uma ordem hierárquica começaria pelo reitor, os chefes de departamento em seguida, então os professores, secretários, zeladores e, por fim, os estudantes. A necessidade de levar em consideração fatores externos leva ao **escalonamento por prioridades**. A ideia básica é direta: a cada processo é designada uma prioridade, e o processo executável com a prioridade mais alta é autorizado a executar.

Mesmo em um PC com um único proprietário, pode haver múltiplos processos, alguns dos quais são mais importantes do que os outros. Por exemplo, a um processo daemon enviando mensagens de correio eletrônico no segundo plano deve ser atribuída uma prioridade mais baixa do que a um processo exibindo um filme de vídeo na tela em tempo real.

Para evitar que processos de prioridade mais alta executem indefinidamente, o escalonador talvez diminua a prioridade do processo que está sendo executado em cada tique do relógio (isto é, em cada interrupção do relógio). Se essa ação faz que a prioridade caia abaixo daquela do próximo processo com a prioridade mais alta, ocorre um chaveamento de processo. Como alternativa, pode ser designado a cada processo um quantum de tempo máximo no qual ele é autorizado a executar. Quando esse quantum for esgotado, o processo seguinte na escala de prioridade recebe uma chance de ser executado.

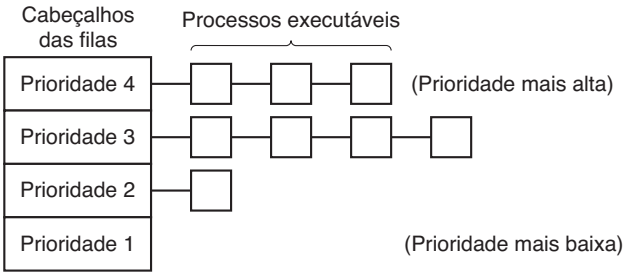
Prioridades podem ser designadas a processos estaticamente ou dinamicamente. Em um computador militar, processos iniciados por generais podem começar com uma prioridade 100, processos iniciados por coronéis a 90, majores a 80, capitães a 70, tenentes a 60 e assim por diante. Como alternativa, em uma central de computação comercial, tarefas de alta prioridade podem custar US$

100 uma hora, prioridade média US$ 75 e baixa prioridade de US$ 50. O sistema UNIX tem um comando, *nice*, que permite que um usuário reduza voluntariamente a prioridade do seu processo, a fim de ser legal com os outros usuários, mas ninguém nunca o utiliza.

Prioridades também podem ser designadas dinamicamente pelo sistema para alcançar determinadas metas. Por exemplo, alguns processos são altamente limitados pela E/S e passam a maior parte do tempo esperando para a E/S ser concluída. Sempre que um processo assim quer a CPU, ele deve recebê-la imediatamente, para deixá-lo iniciar sua próxima solicitação de E/S, que pode então proceder em paralelo com outro processo que estiver de fato computando. Fazer que o processo limitado pela E/S espere muito tempo pela CPU significará apenas tê-lo ocupando a memória por um tempo desnecessariamente longo. Um algoritmo simples para proporcionar um bom serviço para processos limitados pela E/S é configurar a prioridade para $1/f$, onde f é a fração do último quantum que o processo usou. Um processo que usou apenas 1 ms do seu quantum de 50 ms receberia a prioridade 50, enquanto um que usasse 25 ms antes de bloquear receberia a prioridade 2, e um que usasse o quantum inteiro receberia a prioridade 1.

Muitas vezes é conveniente agrupar processos em classes de prioridade e usar o escalonamento de prioridades entre as classes, mas escalonamento circular dentro de cada classe. A Figura 2.43 mostra um sistema com quatro classes de prioridade. O algoritmo de escalonamento funciona do seguinte modo: desde que existam processos executáveis na classe de prioridade 4, apenas execute cada um por um quantum, estilo circular, e jamais se importe com classes de prioridade mais baixa. Se a classe de prioridade 4 estiver vazia, então execute os processos de classe 3 de maneira circular. Se ambas as classes — 4 e 3 — estiverem vazias, então execute a classe 2 de maneira circular e assim por diante. Se as prioridades não forem ajustadas ocasionalmente, classes de prioridade mais baixa podem todas morrer famintas.

FIGURA 2.43 Um algoritmo de escalonamento com quatro classes de prioridade.



Múltiplas filas

Um dos primeiros escalonadores de prioridade foi em CTSS, o sistema compatível de tempo compartilhado do MIT que operava no IBM 7094 (CORBATÓ et al., 1962). O CTSS tinha o problema que o chaveamento de processo era lento, pois o 7094 conseguia armazenar apenas um processo na memória. Cada chaveamento significava trocar o processo atual para o disco e ler em um novo a partir do disco. Os projetistas do CTSS logo perceberam que era mais eficiente dar aos processos limitados pela CPU um grande quantum de vez em quando, em vez de dar a eles pequenos quanta frequentemente (para reduzir as operações de troca). Por outro lado, dar a todos os processos um grande quantum significaria um tempo de resposta ruim, como já vimos. A solução foi estabelecer classes de prioridade. Processos na classe mais alta seriam executados por dois quanta. Processos na classe seguinte seriam executados por quatro quanta etc. Sempre que um processo consumia todos os quanta alocados para ele, era movido para uma classe inferior.

Como exemplo, considere um processo que precisasse computar continuamente por 100 quanta. De início ele receberia um quantum, então seria trocado. Da vez seguinte, ele receberia dois quanta antes de ser trocado. Em sucessivas execuções ele receberia 4, 8, 16, 32 e 64 quanta, embora ele tivesse usado apenas 37 dos 64 quanta finais para completar o trabalho. Apenas 7 trocas seriam necessárias (incluindo a carga inicial) em vez de 100 com um algoritmo circular puro. Além disso, à medida que o processo se aprofundasse nas filas de prioridade, ele seria usado de maneira cada vez menos frequente, poupando a CPU para processos interativos curtos.

A política a seguir foi adotada a fim de evitar punir para sempre um processo que precisasse ser executado por um longo tempo quando fosse iniciado pela primeira vez, mas se tornasse interativo mais tarde. Sempre que a tecla *Enter* era digitada em um terminal, o processo pertencente àquele terminal era movido para a classe de prioridade mais alta, pressupondo que ele estava prestes a tornar-se interativo. Um belo dia, algum usuário com um processo pesadamente limitado pela CPU descobriu que apenas sentar em um terminal e digitar a tecla *Enter* ao acaso de tempos em tempos ajudava e muito seu tempo de resposta. Moral da história: acertar na prática é muito mais difícil que acertar na regra.

Processo mais curto em seguida

Como a *tarefa mais curta primeiro* sempre produz o tempo de resposta médio mínimo para sistemas em

lote, seria bom se ela pudesse ser usada para processos interativos também. Até certo ponto, ela pode ser. Processos interativos geralmente seguem o padrão de esperar pelo comando, executar o comando, esperar pelo comando, executar o comando etc. Se considerarmos a execução de cada comando uma "tarefa" em separado, então podemos minimizar o tempo de resposta geral executando a tarefa mais curta primeiro. O problema é descobrir qual dos processos atualmente executáveis é o mais curto.

Uma abordagem é fazer estimativas baseadas no comportamento passado e executar o processo com o tempo de execução estimado mais curto. Suponha que o tempo estimado por comando para alguns processos é T_0. Agora suponha que a execução seguinte é mensurada como sendo T_1. Poderíamos atualizar nossa estimativa tomando a soma ponderada desses dois números, isto é, $aT_0 + (1 - a)T_1$. Pela escolha de a podemos decidir que o processo de estimativa esqueça as execuções anteriores rapidamente, ou as lembre por um longo tempo. Com $a = 1/2$, temos estimativas sucessivas de

$$T_0,\ T_0/2 + T_1/2,\ T_0/4 + T_1/4 + T_2/2,\ T_0/8 + T_1/8 + T_2/4 + T_3/2$$

Após três novas execuções, o peso de T_0 na nova estimativa caiu para 1/8.

A técnica de estimar o valor seguinte em uma série tomando a média ponderada do valor mensurado atual e a estimativa anterior é às vezes chamada de **envelhecimento** (**aging**). Ela é aplicável a muitas situações onde uma previsão precisa ser feita baseada nos valores anteriores. O envelhecimento é especialmente fácil de implementar quando $a = 1/2$. Tudo o que é preciso fazer é adicionar o novo valor à estimativa atual e dividir a soma por 2 (deslocando-a 1 bit para a direita).

Escalonamento garantido

Uma abordagem completamente diferente para o escalonamento é fazer promessas reais para os usuários a respeito do desempenho e então cumpri-las. Uma promessa realista de se fazer e fácil de cumprir é a seguinte: se n usuários estão conectados enquanto você está trabalhando, você receberá em torno de $1/n$ da potência da CPU. De modo similar, em um sistema de usuário único com n processos sendo executados, todos os fatores permanecendo os mesmos, cada um deve receber $1/n$ dos ciclos da CPU. Isso parece bastante justo.

Para cumprir essa promessa, o sistema deve controlar quanta CPU cada processo teve desde sua criação. Ele então calcula o montante de CPU a que cada um tem direito, especificamente, o tempo desde a criação dividido por n. Tendo em vista que o montante de tempo da CPU que cada processo realmente teve também é conhecido, calcular o índice de tempo de CPU real consumido com o tempo de CPU ao qual ele tem direito é algo bastante direto. Um índice de 0,5 significa que o processo teve apenas metade do que deveria, e um índice de 2,0 significa que teve duas vezes o montante de tempo ao qual ele tinha direito. O algoritmo então executará o processo com o índice mais baixo até que seu índice aumente e se aproxime do de seu competidor. Então este é escolhido para executar em seguida.

Escalonamento por loteria

Embora realizar promessas para os usuários e cumpri-las seja uma bela ideia, ela é difícil de implementar. No entanto, outro algoritmo pode ser usado para gerar resultados similarmente previsíveis com uma implementação muito mais simples. Ele é chamado de **escalonamento por loteria** (WALDSPURGER e WEIHL, 1994).

A ideia básica é dar bilhetes de loteria aos processos para vários recursos do sistema, como o tempo da CPU. Sempre que uma decisão de escalonamento tiver de ser feita, um bilhete de loteria será escolhido ao acaso, e o processo com o bilhete fica com o recurso. Quando aplicado ao escalonamento de CPU, o sistema pode realizar um sorteio 50 vezes por segundo, com cada vencedor recebendo 20 ms de tempo da CPU como prêmio.

Parafraseando George Orwell: "Todos os processos são iguais, mas alguns processos são mais iguais". Processos mais importantes podem receber bilhetes extras, para aumentar a chance de vencer. Se há 100 bilhetes emitidos e um processo tem 20 deles, ele terá uma chance de 20% de vencer cada sorteio. A longo prazo, ele terá acesso a cerca de 20% da CPU. Em comparação com o escalonador de prioridade, em que é muito difícil de afirmar o que realmente significa ter uma prioridade de 40, aqui a regra é clara: um processo que tenha uma fração f dos bilhetes terá aproximadamente uma fração f do recurso em questão.

O escalonamento de loteria tem várias propriedades interessantes. Por exemplo, se um novo processo aparece e ele ganha alguns bilhetes, no sorteio seguinte ele teria uma chance de vencer na proporção do número de bilhetes que tem em mãos. Em outras palavras, o escalonamento de loteria é altamente responsivo.

Processos cooperativos podem trocar bilhetes se assim quiserem. Por exemplo, quando um processo

cliente envia uma mensagem para um processo servidor e então bloqueia, ele pode dar todos os seus bilhetes para o servidor a fim de aumentar a chance de que o servidor seja executado em seguida. Quando o servidor tiver concluído, ele devolve os bilhetes de maneira que o cliente possa executar novamente. Na realidade, na ausência de clientes, os servidores não precisam de bilhete algum.

O escalonamento de loteria pode ser usado para solucionar problemas difíceis de lidar com outros métodos. Um exemplo é um servidor de vídeo no qual vários processos estão alimentando fluxos de vídeo para seus clientes, mas em diferentes taxas de apresentação dos quadros. Suponha que os processos precisem de quadros a 10, 20 e 25 quadros/s. Ao alocar para esses processos 10, 20 e 25 bilhetes, nessa ordem, eles automaticamente dividirão a CPU em mais ou menos a proporção correta, isto é, 10 : 20 : 25.

Escalonamento por fração justa

Até agora presumimos que cada processo é escalonado por si próprio, sem levar em consideração quem é o seu dono. Como resultado, se o usuário 1 inicia nove processos e o usuário 2 inicia um processo, com chaveamento circular ou com prioridades iguais, o usuário 1 receberá 90% da CPU e o usuário 2 apenas 10% dela.

Para evitar essa situação, alguns sistemas levam em conta qual usuário é dono de um processo antes de escaloná-lo. Nesse modelo, a cada usuário é alocada alguma fração da CPU e o escalonador escolhe processos de uma maneira que garanta essa fração. Desse modo, se dois usuários têm cada um 50% da CPU prometidos, cada um receberá isso, não importa quantos processos eles tenham em existência.

Como exemplo, considere um sistema com dois usuários, cada um tendo a promessa de 50% da CPU. O usuário 1 tem quatro processos, *A, B, C* e *D*, e o usuário 2 tem apenas um processo, *E*. Se o escalonamento circular for usado, uma sequência de escalonamento possível que atende a todas as restrições é a seguinte:

A E B E C E D E A E B E C E D E ...

Por outro lado, se o usuário 1 tem direito a duas vezes o tempo de CPU que o usuário 2, talvez tenhamos

A B E C D E A B E C D E ...

Existem numerosas outras possibilidades, é claro, e elas podem ser exploradas, dependendo de qual seja a noção de justiça.

2.4.4 Escalonamento em sistemas de tempo real

Um sistema de **tempo real** é aquele em que o tempo tem um papel essencial. Tipicamente, um ou mais dispositivos físicos externos ao computador geram estímulos, e o computador tem de reagir em conformidade dentro de um montante de tempo fixo. Por exemplo, o computador em um CD player recebe os bits à medida que eles saem do drive e deve convertê-los em música dentro de um intervalo muito estrito. Se o cálculo levar tempo demais, a música soará estranha. Outros sistemas de tempo real estão monitorando pacientes em uma UTI, o piloto automático em um avião e o controle de robôs em uma fábrica automatizada. Em todos esses casos, ter a resposta certa, mas tê-la tarde demais é muitas vezes tão ruim quanto não tê-la.

Sistemas em tempo real são geralmente categorizados como **tempo real crítico**, significando que há prazos absolutos que devem ser cumpridos — para valer! — e **tempo real não crítico**, significando que descumprir um prazo ocasional é indesejável, mas mesmo assim tolerável. Em ambos os casos, o comportamento em tempo real é conseguido dividindo o programa em uma série de processos, cada um dos quais é previsível e conhecido antecipadamente. Esses processos geralmente têm vida curta e podem ser concluídos em bem menos de um segundo. Quando um evento externo é detectado, cabe ao escalonador programar os processos de uma maneira que todos os prazos sejam atendidos.

Os eventos a que um sistema de tempo real talvez tenha de responder podem ser categorizados ainda como **periódicos** (significando que eles ocorrem em intervalos regulares) ou **aperiódicos** (significando que eles ocorrem de maneira imprevisível). Um sistema pode ter de responder a múltiplos fluxos de eventos periódicos. Dependendo de quanto tempo cada evento exige para o processamento, tratar de todos talvez não seja nem possível. Por exemplo, se há m eventos periódicos e o evento i ocorre com o período P_i e exige C_i segundos de tempo da CPU para lidar com cada evento, então a carga só pode ser tratada se

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Diz-se de um sistema de tempo real que atende a esse critério que ele é **escalonável**. Isso significa que ele realmente pode ser implementado. Um processo que fracassa em atender esse teste não pode ser escalonado, pois o montante total de tempo de CPU que os processos querem coletivamente é maior do que a CPU pode proporcionar.

Como exemplo, considere um sistema de tempo real não crítico com três eventos periódicos, com períodos de 100, 200 e 500 ms, respectivamente. Se esses eventos exigem 50, 30 e 100 ms de tempo da CPU, respectivamente, o sistema é escalonável, pois 0,5 + 0,15 + 0,2 < 1. Se um quarto evento com um período de 1 segundo é acrescentado, o sistema permanecerá escalonável desde que esse evento não precise de mais de 150 ms de tempo da CPU por evento. Implícito nesse cálculo está o pressuposto de que o overhead de chaveamento de contexto é tão pequeno que pode ser ignorado.

Algoritmos de escalonamento de tempo real podem ser estáticos ou dinâmicos. Os primeiros tomam suas decisões de escalonamento antes de o sistema começar a ser executado. Os últimos tomam suas decisões no tempo de execução, após ela ter começado. O escalonamento estático funciona apenas quando há uma informação perfeita disponível antecipadamente sobre o trabalho a ser feito, e os prazos que precisam ser cumpridos. Algoritmos de escalonamento dinâmico não têm essas restrições.

2.4.5 Política *versus* mecanismo

Até o momento, presumimos tacitamente que todos os processos no sistema pertencem a usuários diferentes e estão, portanto, competindo pela CPU. Embora isso seja muitas vezes verdadeiro, às vezes acontece de um processo ter muitos filhos executando sob o seu controle. Por exemplo, um processo de sistema de gerenciamento de banco de dados pode ter muitos filhos. Cada filho pode estar funcionando em uma solicitação diferente, ou cada um pode ter alguma função específica para realizar (análise sintática de consultas, acesso ao disco etc.). É inteiramente possível que o principal processo tenha uma ideia excelente de qual dos filhos é o mais importante (ou tenha tempo crítico) e qual é o menos importante. Infelizmente, nenhum dos escalonadores discutidos aceita qualquer entrada dos processos do usuário sobre decisões de escalonamento. Como resultado, o escalonador raramente faz a melhor escolha.

A solução desse problema é separar o **mecanismo de escalonamento** da **política de escalonamento**, um princípio há muito estabelecido (LEVIN et al., 1975). O que isso significa é que o algoritmo de escalonamento é parametrizado de alguma maneira, mas os parâmetros podem estar preenchidos pelos processos dos usuários. Vamos considerar o exemplo do banco de dados novamente. Suponha que o núcleo utilize um algoritmo de escalonamento de prioridades, mas fornece uma chamada de sistemas pela qual um processo pode estabelecer (e mudar) as prioridades dos seus filhos. Dessa maneira, o pai pode controlar como seus filhos são escalonados, mesmo que ele mesmo não realize o escalonamento. Aqui o mecanismo está no núcleo, mas a política é estabelecida por um processo do usuário. A separação do mecanismo de política é uma ideia fundamental.

2.4.6 Escalonamento de threads

Quando vários processos têm cada um múltiplos threads, temos dois níveis de paralelismo presentes: processos e threads. Escalonar nesses sistemas difere substancialmente, dependendo se os threads de usuário ou os threads de núcleo (ou ambos) recebem suporte.

Vamos considerar primeiro os threads de usuário. Tendo em vista que o núcleo não tem ciência da existência dos threads, ele opera como sempre fez, escolhendo um processo, digamos, *A*, e dando a *A* controle de seu quantum. O escalonador de thread dentro de *A* decide qual thread executar, digamos, *A1*. Dado que não há interrupções de relógio para multiprogramar threads, esse thread pode continuar a ser executado por quanto tempo quiser. Se ele utilizar todo o quantum do processo, o núcleo selecionará outro processo para executar.

Quando o processo *A,* por fim, executar novamente, o thread *A1* retomará a execução. Ele continuará a consumir todo o tempo de *A* até que termine. No entanto, seu comportamento antissocial não afetará outros processos. Eles receberão o que quer que o escalonador considere sua fração apropriada, não importa o que estiver acontecendo dentro do processo *A*.

Agora considere o caso em que os threads de *A* tenham relativamente pouco trabalho para fazer por surto de CPU, por exemplo, 5 ms de trabalho dentro de um quantum de 50 ms. Em consequência, cada um executa por um tempo, então cede a CPU de volta para o escalonador de threads. Isso pode levar à sequência *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, antes que o núcleo chaveie para o processo *B*. Essa situação está mostrada na Figura 2.44(a).

O algoritmo de escalonamento usado pelo sistema de tempo de execução pode ser qualquer um dos descritos anteriormente. Na prática, o escalonamento circular e o de prioridade são os mais comuns. A única restrição é a ausência de um relógio para interromper um thread que esteja sendo executado há tempo demais. Visto que os threads cooperam, isso normalmente não é um problema.

Agora considere a situação com threads de núcleo. Aqui o núcleo escolhe um thread em particular para executar. Ele não precisa levar em conta a qual processo o thread pertence, porém ele pode, se assim o desejar. O thread recebe um quantum e é suspenso compulsoriamente se o exceder. Com um quantum de 50 ms, mas threads que são