Universidade Federal Rural de Pernambuco
Sistemas de Informação

Princípios de Software Básico

Lhaíslla Eduarda Cavalcanti Rodrigues da Silva

RECIFE-PE

Exercícios referente ao capítulo 2- Processos e Threads e Deadlocks do livro Sistemas Operacionais modernos Andrew . S Tanenbaum

1. Na Figura 2.2, são mostrados três estados de processos. Na teoria, com três estados, poderia haver seis transições, duas para cada. No entanto, apenas quatro transições são mostradas. Existe alguma circunstância na qual uma delas ou ambas as transições perdidas possam ocorrer?

   As duas transições que faltam no diagrama de estados são “bloqueado para em execução”e de “pronto para bloqueado”. Na primeira opção, supondo que um processo de entrada e saída de um dispositivo esteja bloqueado e esse processo termina, caso a CP U esteja ociosa, esta transição poderia acontecer. Já a segunda opção não é possível de acontecer, já que somente os processos em execução podem ser bloqueados.

2. Suponha que você fosse projetar uma arquitetura de computador avançada que realizasse chaveamento de processos em hardware, em vez de interrupções. De qual informação a CPU precisaria? Descreva como o processo de chaveamento por hardware poderia funcionar.

   A CPU precisaria do tempo de iniciação e termino dos processos, então diferente do processo de alternância não precisaria finalizar um processo para que o outro pudesse ser executado o processo de alternância por hardware tem a capacidade de executar dois processos ao mesmo tempo.

3. Em todos os computadores atuais, pelo menos parte dos tratadores de interrupções é escrita em linguagem de montagem. Por quê?

   Porque ações como salvar os registradores e alterar o ponteiro de pilha não podem ser expressas em linguagens de alto nível como C, assim elas são implementadas por uma pequena rotina em linguagem de montagem.

4. Quando uma interrupção ou uma chamada de sistema transfere controle para o sistema operacional, geralmente uma área da pilha do núcleo separada da pilha do processo interrompido é usada. Por quê?

   Para identificar os processos filhos que dependem deste processo e o tempo de finalização.

5. Um sistema computacional tem espaço suficiente para conter cinco programas em sua memória principal. Esses programas estão ociosos esperando por E/S metade do tempo. Qual fração do tempo da CPU é desperdiçada?

   A fração de tempo desperdiçada da CPU seria de 0,5^5 = 0,03125 ou 1/32 correspondente à chance de todos os cinco processos estarem ociosos.

6. Um computador tem 4 GB de RAM da qual o sistema operacional ocupa 512 MB. Os processos ocupam 256 MB cada (para simplificar) e têm as mesmas características. Se a meta é a utilização de 99% da CPU, qual é a espera de E/S máxima que pode ser tolerada?

   512 - 256 MB / 4 = 64 % 99 = 63,36

7. Múltiplas tarefas podem ser executadas em paralelo e terminar mais rápido do que se forem executadas de modo sequencial. Suponha que duas tarefas, cada uma precisando de 20 minutos de tempo da CPU, iniciassem simultaneamente. Quanto tempo a última levará para completar se forem executadas sequencialmente? Quanto tempo se forem executadas em paralelo? Presuma uma espera de E/S de 50%.

   Presumindo que o tempo de espera de E/S é de 50%, a últ ima tarefa ter minará aos 60 minutos caso seja executada sequencialmente. Isso porque como cada tarefa corresponde a 50% de tempo da C PU (10 minutos x 2 tarefas = 20 minutos de espera para cada execução de tarefa), logo, se cada uma for execut ada sequencialmente seria 30 minutos para a o t érmino da prime ira tarefa, e somente após esse tempo decorrido a segunda tarefa seria inicializada, terminando sua execução com 60 minutos. Para o caso da execução em paralelo, o tempo gasto seria de 30 minutos, já que as duas tarefas são executadas simultaneamente em 30 minut os, onde 20 mi nutos desse tempo correspondem ao tempo da CPU e os outros 10 minutos correspondem ao tempo de espera

8. Considere um sistema multiprogramado com grau de 6 (isto é, seis programas na memória ao mesmo tempo). Presuma que cada processo passe 40% do seu tempo esperando pelo dispositivo de E/S. Qual será a utilização da CPU?

   A utilização da CPU é calculada através da fórmula 1-(p^n), onde p é a fração de tempo pela espera de uma entrada ou saída de um dispositivo e corresponde ao número de processos. Logo, a chance de que todos os processos estejam à espera de entrada e saída de um dispositivo é 1-(0,4^6) = 1-0,004096 =

0,995904, ou seja, aproximadamente 99,6% de utilização da CPU.

9. Presuma que você esteja tentando baixar um arquivo grande de 2 GB da internet. O arquivo está disponível a partir de um conjunto de servidores espelho, cada um deles capaz de fornecer um subconjunto dos bytes do arquivo; presuma que uma determinada solicitação especifique os bytes de início e fim do arquivo. Explique como você poderia usar os threads para melhorar o tempo de download

   O s threads poderiam ser utilizados para gerenciar os downloads de arquivos , fazendo com que ess es arquivos fossem divididos em partes , onde cada therad ficaria responsável por um determinado pedaço do arquivo, ass im o download seria feito d e forma conjunta sem perda de velocidade.

10. No texto foi afirmado que o modelo da Figura 2.11(a) não era adequado a um servidor de arquivos usando um cache na memória. Por que não? Será que cada processo poderia ter seu próprio cache?

    Pois cada um dos núcleos pode executar apenas um processo de cada vez. 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.

11. Se um processo multithread bifurca, um problema ocorre se o filho recebe cópias de todos os threads do pai. Suponha que um dos threads originais estivesse esperando por entradas do teclado. Agora dois threads estão esperando por entradas do teclado, um em cada processo. Esse problema ocorre alguma vez em processos de thread único?

    Não. Se um processo de thread único estiver bloqueado no teclado, ele não poderá bifurcar-se.

12. Um servidor web multithread é mostrado na Figura 2.8. Se a única maneira de ler de um arquivo é a chamada de sistema read com bloqueio normal, você acredita que threads de usuário ou threads de núcleo estão sendo usados para o servidor web? Por quê?
    a. Um segmento de trabalho será bloqueado quando for necessário ler uma página da Web do disco. Se estiverem sendo usados encadeamentos no nível do usuário, essa ação bloqueará todo o processo, destruindo o valor do multithreading. Portanto, é essencial que os threads do kernel sejam usados para permitir que alguns threads sejam bloqueados sem afetar os outros.

13. No texto, descrevemos um servidor web multithread, mostrando por que ele é melhor do que um servidor de thread único e um servidor de máquina de estado finito. Existe alguma circunstância na qual um servidor de thread único possa ser melhor? Dê um exemplo.
    a. Sim. Se o servidor estiver totalmente vinculado à CPU, não haverá necessidade de vários threads. Isso apenas adiciona complexidade

desnecessária. Como exemplo, considere um número de assistência da lista telefônica (como 555-1212) para uma área com 1 milhão de pessoas. Se cada registro (nome, número de telefone) tiver, digamos, 64 caracteres, o banco de dados inteiro ocupará 64 megabytes e poderá ser facilmente mantido na memória do servidor para fornecer uma pesquisa rápida.

14. Na Figura 2.12, o conjunto de registradores é listado como um item por thread em vez de por processo. Por quê? Afinal de contas, a máquina tem apenas um conjunto de registradores.
    a. Quando um encadeamento é parado, ele possui valores nos registradores. Eles devem ser salvos, exatamente como quando o processo é interrompido, os registros devem ser salvos. A multiprogramação de threads não é diferente dos processos de multiprogramação, portanto, cada thread precisa de sua própria área de salvamento de registro.

15. Por que um thread em algum momento abriria mão voluntariamente da CPU chamando thread_yield? Afinal, visto que não há uma interrupção periódica de relógio, ele talvez jamais receba a CPU de volta.
    a. A função thread_yield permite que um thread abra mão voluntariamente da CPU para que outro thread possa ser executado. Uma chamada dessas é importante porque não há uma interrupção de relógio para realmente forçar a multiprogramação como há com os processos.Assim, é fundamental que os que os threads sejam educados e voluntariamente entregam a CPU de tempos em tempos para dar aos outros threads uma chance de serem executados.

16. É possível que um thread seja antecipado por uma interrupção de relógio? Se a resposta for afirmativa, em quais circunstâncias?
    a. Sim, a thread_yield, que permite que um thread abra mão voluntariamente da CPU para deixar outro thread ser executado. Uma chamada dessas é importante porque não há uma interrupção de relógio para realmente forçar a multiprogramação como há com os processos.

17. Neste problema, você deve comparar a leitura de um arquivo usando um servidor de arquivos de um thread único e um servidor com múltiplos threads. São necessários 12 ms para obter uma requisição de trabalho, despachá- -la e realizar o resto do processamento, presumindo que os dados necessários estejam na cache de blocos. Se uma operação de disco for necessária, como é o caso em um terço das vezes, 75 ms adicionais são necessários, tempo em que o thread repousa. Quantas requisições/segundo o servidor consegue lidar se ele tiver um único thread? E se ele for multithread?
    a. No caso de thread único, os acertos do cache levam 12 ms e os erros do cache, 87 ms. A média ponderada é $2/3 \times 12 + 1/3 \times 87$. Portanto, a solicitação média leva 37 ms e o servidor pode executar cerca de 27 por segundo. Para um servidor multithread, toda a espera do disco é sobreposta, portanto, toda solicitação leva 12 ms e o servidor pode lidar com 83 solicitações de 1/3 por segundo.

18. Qual é a maior vantagem de se implementar threads no espaço de usuário? Qual é a maior desvantagem?
    a. A maior vantagem é a eficiência. Não são necessários traps no kernel para alternar threads. A maior desvantagem é que, se um thread é bloqueado, todo o processo é bloqueado.

19. . Na Figura 2.15 as criações dos thread e mensagens impressas pelos threads são intercaladas ao acaso. Existe alguma maneira de se forçar que a ordem seja estritamente thread 1 criado, thread 1 imprime mensagem, thread 1 sai, thread 2 criado, thread 2 imprime mensagem, thread 2 sai e assim por diante? Se a resposta for afirmativa, como? Se não, por que não?
    a. Sim, isso pode ser feito. Após cada chamada para criar o pthread, o programa principal pode fazer uma junção do pthread para aguardar a saída do encadeamento recém-criado antes de criar o próximo encadeamento.
20. Na discussão sobre variáveis globais em threads, usamos uma rotina create_global para alocar memória para um ponteiro para a variável, em vez de para a própria variável. Isso é essencial, ou as rotinas poderiam funcionar somente com os próprios valores?
    a. Os ponteiros são realmente necessários porque o tamanho da variável global é desconhecido.Pode ser qualquer coisa, de um caractere a uma série de números de ponto flutuante. Se o valor fosse armazenado, seria necessário fornecer o tamanho para criar global, o que é correto.

21. Considere um sistema no qual threads são implementados inteiramente no espaço do usuário, com o sistema de tempo de execução sofrendo uma interrupção de relógio a cada segundo. Suponha que uma interrupção de relógio ocorra enquanto um thread está executando no sistema de tempo de execução. Qual problema poderia ocorrer? Você poderia sugerir uma maneira para solucioná-lo?
    a. Poderia ocorrer problema relacionado ao compartilhamento de estruturas de dados.Suponha que um thread observe que há pouca memória e comece a alocar mais memória. No meio do caminho há um chaveamento de threads, e o novo também observa que há pouca memória e também começa a alocar mais memória. A memória provavelmente será alocada duas vezes. Esses problemas podem ser solucionados com algum esforço, mas os programas de multithread devem ser pensados e projetados com cuidado para funcionarem corretamente.

22. Suponha que um sistema operacional não tem nada parecido com a chamada de sistema select para saber antecipadamente se é seguro ler de um arquivo, pipe ou dispositivo, mas ele permite que relógios de alarme sejam configurados para interromper chamadas de sistema bloqueadas. É possível implementar um pacote de threads no espaço do usuário nessas condições? Discuta.
    a. Uma solução possível para o problema de threads sendo executados infinitamente é obrigar o sistema de tempo de execução a solicitar um sinal de relógio (interrupção) a cada segundo para dar a ele o controle, mas isso, também, é algo grosseiro e confuso para o programa. Interrupções periódicas de relógio em uma frequência mais alta nem sempre são possíveis, e mesmo que fossem, a sobrecarga total poderia ser substancial. Além disso, um thread talvez precise também de uma interrupção de relógio, interferindo com o uso

do relógio pelo sistema de tempo de execução.

23. A solução da espera ocupada usando a variável turn (Figura 2.23) funciona quando os dois processos estão executando em um multiprocessador de memória compartilhada, isto é, duas CPUs compartilhando uma memória comum?
    a. Sim, ainda funciona, mas ainda está ocupado esperando, é claro, essa solução exige que os dois processos alternem-se estritamente na entrada em suas regiões críticas.

24. A solução de Peterson para o problema da exclusão mútua mostrado na Figura 2.24 funciona quando o escalonamento de processos é preemptivo? E quando ele é não preemptivo?
    a. Certamente funciona com agendamento preventivo. De fato, ele foi projetado para esse caso. Quando o planejamento não é preventivo, ele pode falhar. Considere o caso em que o turno é inicialmente 0, mas o processo 1 é executado primeiro. Ele fará um loop para sempre e nunca liberará a CPU.

25. O problema da inversão de prioridades discutido na Seção 2.3.4 acontece com threads de usuário? Por que ou por que não?
    a. O problema de inversão de prioridade ocorre quando um processo de baixa prioridade está em sua região crítica e de repente um processo de alta prioridade fica pronto e é agendado. Se ele usa a espera ocupada, ele será executado para sempre. Nos encadeamentos no nível do usuário, não é possível que um encadeamento de baixa prioridade seja subitamente antecipado para permitir a execução de um encadeamento de alta prioridade. Não há preempção. Com os threads no nível do kernel, esse problema pode surgir.

26. Na Seção 2.3.4, uma situação com um processo de alta prioridade, H, e um processo de baixa prioridade, L, foi descrita, o que levou H a entrar em um laço infinito. O mesmo problema ocorre se o escalonamento circular for usado em vez do escalonamento de prioridade? Discuta.
    a. Quando um processo quer entrar em sua região crítica, ele confere para ver se a entrada é permitida. Se não for, o processo apenas esperará em um laço apertado até que isso seja permitido. Não apenas essa abordagem desperdiça tempo da CPU, como também pode ter efeitos inesperados. Considere um computador com dois processos, H, com alta prioridade, e L, com baixa prioridade. As regras de escalonamento são colocadas de tal forma que H é executado sempre que ele estiver em um estado pronto. Em um determinado momento, com L em sua região crítica, H torna-se pronto para executar (por exemplo, completa uma operação de E/S). H agora começa a espera ocupada, mas tendo em vista que L nunca é escalonado enquanto H estiver executando, L jamais recebe a chance de deixar a sua região crítica, de maneira que H segue em um laço infinito. Essa situação às vezes é referida como problema da inversão de prioridade.

27. Em um sistema com threads, há uma pilha por thread ou uma pilha por processo quando threads de usuário são usados? E quando threads de núcleo são usados? Explique.

a. Cada encadeamento chama procedimentos por conta própria, portanto deve ter sua própria pilha para as variáveis locais, endereços de retorno e assim por diante. Isso é verdade tanto para threads no nível do usuário quanto para threads no nível do kernel.

28. Quando um computador está sendo desenvolvido, normalmente ele é primeiro simulado por um programa que executa uma instrução de cada vez. Mesmo multiprocessadores são simulados de maneira estritamente sequencial. É possível que uma condição de corrida ocorra quando não há eventos simultâneos como nesses casos?
    a. Sim. O computador simulado pode ser multiprogramado. Por exemplo, enquanto o processo A está em execução, ele lê alguma variável compartilhada. Então, um tique de relógio simulado acontece e o processo B é executado. Ele também lê a mesma variável. Em seguida, adiciona 1 à variável. Quando o processo A é executado, se também adicionar 1 à variável, temos uma condição de corrida.

29. O problema produtor-consumidor pode ser ampliado para um sistema com múltiplos produtores e consumidores que escrevem (ou leem) para (ou de) um buffer compartilhado. Presuma que cada produtor e consumidor executem seu próprio thread. A solução apresentada na Figura 2.28 usando semáforos funcionaria para esse sistema?
    a. Sim, funcionará como está. Em um dado instante, apenas um produtor (consumidor) pode adicionar (remover) um item ao (do) buffer.

30. Considere a solução a seguir para o problema da exclusão mútua envolvendo dois processos P0 e P1. Presuma que a variável turn seja inicializada para 0. O código do processo P0 é apresentado a seguir. /* Outro codigo */ while (turn != 0) { } /* Nao fazer nada e esperar */ Critical Section /* . . . */ turn = 0; /* Outro codigo */ Para o processo P1, substitua 0 por 1 no código anterior. Determine se a solução atende a todas as condições exigidas para uma solução de exclusão mútua.
    a. A solução satisfaz a exclusão mútua, pois não é possível que ambos os processos estejam em sua seção crítica. Ou seja, quando o turn é 0, P0 pode executar sua seção crítica, mas não P1. Da mesma forma, quando o turn é 1. No entanto, isso pressupõe que P0 deve ser executado primeiro. Se P1 produz algo e o coloca em um buffer, enquanto P0 pode entrar em sua seção crítica, ele encontrará o buffer vazio e bloqueado. Além disso, essa solução exige uma alternância rigorosa dos dois processos, o que é indesejável.

31. Como um sistema operacional capaz de desabilitar interrupções poderia implementar semáforos
    a. Para fazer uma operação de semáforo, o sistema operacional primeiro desativa as interrupções. Em seguida, ele lê o valor do semáforo. Se estiver fazendo um down e o semáforo for igual a zero, colocará o processo de chamada em uma lista de processos bloqueados associados ao semáforo. Se estiver fazendo um up, deve verificar se algum processo está bloqueado no semáforo. Se um ou mais processos estiverem bloqueados, um deles será removido da lista de processos bloqueados e tornado executável. Quando todas essas operações forem concluídas, as interrupções poderão ser ativadas novamente.

32. Mostre como semáforos contadores (isto é, semáforos que podem armazenar um valor arbitrário) podem ser implementados usando apenas semáforos binários e instruções de máquinas ordinárias.
    a. está um contador que mantém o número de altos menos o número de baixas e uma lista de processos bloqueados nesse semáforo. Para implementar para baixo, um processo obtém primeiro acesso exclusivo aos semáforos, contador e lista fazendo um down em M. Depois, diminui o contador. Se for zero ou mais, apenas aumenta M e sai. Se M for negativo, o processo é colocado na lista de processos bloqueados. Em seguida, é feito um up em M e um down em B para bloquear o processo. Para implementar, o primeiro M é derrubado para obter exclusão mútua e, em seguida, o contador é incrementado. Se for maior que zero, ninguém foi bloqueado; portanto, tudo o que precisa ser feito é aumentar M. Se, no entanto, o contador agora for negativo ou zero, é necessário remover algum processo da lista. Finalmente, um up é feito em B e M nessa ordem.

33. Se um sistema tem apenas dois processos, faz sentido usar uma barreira para sincronizá-los? Por que ou por que não?
    a. Se o programa opera em fases e nenhum processo pode entrar na próxima fase até que ambos sejam finalizados com a fase atual, faz todo o sentido usar uma barreira.

34. É possível que dois threads no mesmo processo sincronizem usando um semáforo do núcleo se os threads são implementados pelo núcleo? E se eles são implementados no espaço do usuário? Presuma que nenhum thread em qualquer outro processo tenha acesso ao semáforo. Discuta suas respostas.
    a. 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. Essa atomicidade é absolutamente essencial para solucionar problemas de sincronização e evitar condições de corrida.

35. A sincronização dentro de monitores usa variáveis de condição e duas operações especiais, wait e signal. Uma forma mais geral de sincronização seria ter uma única primitiva, waituntil, que tivesse um predicado booleano arbitrário como parâmetro. Desse modo, você poderia dizer, por exemplo, waituntil x < 0 ou y + z < n A primitiva signal não seria mais necessária. Esse esquema é claramente mais geral do que o de Hoare ou Brinch Hansen, mas não é usado. Por que não? (Dica: pense a respeito da implementação.)
    a. É muito caro de implementar. Sempre que qualquer variável que aparece em um predicado em que algum processo está aguardando alterações, o sistema de tempo de execução deve reavaliar o predicado para verificar se o processo pode ser desbloqueado. Com os monitores Hoare e Brinch Hansen, os processos só podem ser despertados em um sinal primitivo.

36. Uma lanchonete tem quatro tipos de empregados: (1) atendentes, que pegam os pedidos dos clientes; (2) cozinheiros, que preparam a comida; (3) especialistas em empacotamento, que colocam a comida nas sacolas; e (4) caixas, que dão as sacolas para os clientes e recebem seu dinheiro. Cada empregado pode ser considerado um processo sequencial comunicante. Que forma de comunicação entre processos eles usam? Relacione esse modelo aos processos em UNIX.

a. Os funcionários se comunicam passando mensagens: pedidos, comida e malas neste caso. Em termos UNIX, os quatro processos são conectados por pipes.

37. Suponha que temos um sistema de transmissão de mensagens usando caixas de correio. Quando envia para uma caixa de correio cheia ou tenta receber de uma vazia, um processo não bloqueia. Em vez disso, ele recebe de volta um código de erro. O processo responde ao código de erro apenas tentando de novo, repetidas vezes, até ter sucesso. Esse esquema leva a condições de corrida?
    a. Sim, pois podem compartilhar de alguma memória comum que cada um pode ler e escrever. A memória compartilhada pode encontrar-se na memória principal (possivelmente em uma estrutura de dados de núcleo) ou ser um arquivo compartilhado; o local da memória compartilhada não muda a natureza da comunicação ou os problemas que surgem.

38. Os computadores CDC 6600 poderiam lidar com até 10 processos de E/S simultaneamente usando uma forma interessante de escalonamento circular chamado de compartilhamento de processador. Um chaveamento de processo ocorreu após cada instrução, de maneira que a instrução 1 veio do processo 1, a instrução 2 do processe 2 etc. O chaveamento de processo foi realizado por um hardware especial e a sobrecarga foi zero. Se um processo necessitasse T segundos para completar na ausência da competição, de quanto tempo ele precisaria se o compartilhamento de processador fosse usado com n processos?
    a. O processo precisarioa de 2T ^n caso fosse compartilhado com n processos.

39. Considere o fragmento de código C seguinte: void main( ) { fork( ); fork( ); exit( ); } Quantos processos filhos são criados com a execução desse programa?
    a. São criados quatro processos, pois existe uma bifurcação no primeiro fork(), entre uma cópia do processo pai e a criação de um processo filho, e logo depois c ada um deles se bifurca novamente no fork() seguinte, onde mais dois processos são criados.

40. Escalonadores circulares em geral mantêm uma lista de todos os processos executáveis, com cada processo ocorrendo exatamente uma vez na lista. O que aconteceria se um processo ocorresse duas vezes? Você consegue pensar em qualquer razão para permitir isso?
    a. 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.

41. É possível determinar se um processo é propenso a se tornar limitado pela CPU ou limitado pela E/S analisando o código fonte? Como isso pode ser determinado no tempo de execução?

a. Em casos simples, pode ser possível verificar se a E / S será limitada observando o código-fonte. Por exemplo, um programa que lê todos os seus arquivos de entrada em buffers no início provavelmente não será vinculado à E / S, mas é provável que um problema que leia e grave incrementalmente em vários arquivos diferentes (como um compilador) seja I / O O ligado. Se o sistema operacional fornecer um recurso, como o comando ps do UNIX, que pode informar a quantidade de tempo de CPU usada por um programa, você poderá compará-lo com o tempo total para concluir a execução do programa. Obviamente, isso é mais significativo em um sistema em que você é o único usuário.

42. Explique como o valor quantum de tempo e tempo de chaveamento de contexto afetam um ao outro, em um algoritmo de escalonamento circular.
    a. 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. Quando o processo usa todo o seu quantum, ele é colocado no fim da lista.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.

43. Medidas de um determinado sistema mostraram que o processo típico executa por um tempo T antes de bloquear na E/S. Um chaveamento de processo exige um tempo S, que é efetivamente desperdiçado (sobrecarga). Para o escalonamento circular com quantum Q, dê uma fórmula para a eficiência da CPU para cada uma das situações a seguir: (a) $Q = \infty$. (b) $Q > T$. (c) $S < Q < T$. (d) $Q = S$. (e) Q quase 0.
    a. A eficiência da CPU é o tempo útil da CPU dividido pelo tempo total da CPU. Quando $Q \geq T$, o ciclo básico é que o processo seja executado para T e sofra uma troca de processo para S. Assim, (a) e (b) têm uma eficiência de $T / (S + T)$. Quando o quantum é menor que T, cada execução de T exigirá comutações de processo T / Q, perdendo um tempo ST / Q. A eficiência aqui é então $T/(T+(ST/Q))$ que reduz a $Q / (Q + S)$, que é a resposta para (c). Em (d), apenas substituímos Q por S e descobrimos que a eficiência é de 50%. Finalmente, para (e), como $Q \rightarrow 0$, a eficiência vai para 0.

44. Cinco tarefas estão esperando para serem executadas. Seus tempos de execução esperados são 9, 6, 3, 5 e X. Em qual ordem elas devem ser executadas para minimizar o tempo de resposta médio? (Sua resposta dependerá de X.)
    a. O trabalho mais curto primeiro é o caminho para minimizar o tempo médio de resposta.
    $0<X \leq 3$: X, 3, 5, 6, 9.
    $3<X \leq 5$: 3, X, 5, 6, 9.
    $5<X \leq 6$: 3, 5, X, 6, 9.
    $6<X \leq 9$: 3, 5, 6, X, 9.

X > 9: 3, 5, 6, 9,X

45. Cinco tarefas em lote, A até E, chegam a um centro de computadores quase ao mesmo tempo. Elas têm tempos de execução estimados de 10, 6, 2, 4 e 8 minutos. Suas prioridades (externamente determinadas) são 3, 5, 2, 1 e 4, respectivamente, sendo 5 a mais alta. Para cada um dos algoritmos de escalonamento a seguir, determine o tempo de retorno médio do processo. Ignore a sobrecarga de chaveamento de processo. (a) Circular. (b) Escalonamento por prioridade. (c) Primeiro a chegar, primeiro a ser servido (siga a ordem 10, 6, 2, 4, 8). (d) Tarefa mais curta primeiro. Para (a), presuma que o sistema é multiprogramado e que cada tarefa recebe sua porção justa de tempo na CPU. Para (b) até (d), presuma que apenas uma tarefa de cada vez é executada, até terminar. Todas as tarefas são completamente limitadas pela CPU.
    a. Para rodízio, durante os primeiros 10 minutos, cada trabalho recebe 1/5 da CPU. Ao fim de 10 minutos, o vídeo termina. Durante os próximos 8 minutos, cada trabalho recebe 1/4 da CPU, após o que o término termina. Então, cada um dos três trabalhos restantes obtém 1/3 da CPU por 6 minutos, até que B termine, e assim por diante. Os tempos de finalização dos cinco trabalhos são 10, 18, 24, 28 e 30, por uma média de 22 minutos. Para o agendamento prioritário, B é executado primeiro. Após 6 minutos, está terminado. Os outros trabalhos terminam aos 14, 24, 26 e 30, por uma média de 18,8 minutos. Se os trabalhos são executados na ordem de A a E, terminam em 10, 16, 18, 22 e 30, por uma média de 19,2 minutos. Por fim, o primeiro trabalho mais curto produz tempos de acabamento de 2, 6, 12, 20 e 30, por uma média de 14 minutos.

46. Um processo executando em CTSS precisa de 30 quanta para ser completo. Quantas vezes ele deve ser trocado para execução, incluindo a primeiríssima vez (antes de ter sido executado)?
    a. 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, apenas 4 trocas.

47. Considere um sistema de tempo real com duas chamadas de voz de periodicidade de 5 ms cada com um tempo de CPU por chamada de 1 ms, e um fluxo de vídeo de periodicidade de 33 ms com tempo de CPU por chamada de 11 ms. Esse sistema é escalonável?
    a. Cada chamada de voz precisa de 200 amostras de 1 ms ou 200 ms. Juntos, eles usam 400 ms de tempo de CPU. O vídeo precisa de 11 ms 33 1/3 vezes por segundo para um total de cerca de 367 ms. A soma é de 767 ms por segundo em tempo real, para que o sistema seja agendado.

48. Para o problema 47, será que outro fluxo de vídeo pode ser acrescentado e ter o sistema ainda escalonável?
    a. Outro fluxo de vídeo consome 367 ms de tempo por segundo, totalizando 1134 ms por segundo de tempo real, para que o sistema não seja agendado.
49. O algoritmo de envelhecimento com a = 1/2 está sendo usado para prever tempos de execução. As quatro execuções anteriores, da mais antiga à mais recente, são 40, 20, 40 e 15 ms. Qual é a previsão do próximo tempo?
    a. 40 ms , pois irá chamar o processo pai.

50. Um sistema de tempo real não crítico tem quatro eventos periódicos com períodos de 50, 100, 200 e 250 ms cada. Suponha que os quatro eventos exigem 35, 20, 10 e x ms de tempo da CPU, respectivamente. Qual é o maior valor de x para o qual o sistema é escalonável?
    a. A fração da CPU usada é 35/50 + 20/100 + 10/200 + x / 250. Para ser agendável, isso deve ser menor que 1. Portanto, x deve ser menor que 12,5 ms.

51. No problema do jantar dos filósofos, deixe o protocolo a seguir ser usado: um filósofo de número par sempre pega o seu garfo esquerdo antes de pegar o direito; um filósofo de número ímpar sempre pega o garfo direito antes de pegar o esquerdo. Esse protocolo vai garantir uma operação sem impasse?
    a. Sim. Sempre haverá pelo menos um garfo livre e pelo menos um filósofo que pode obter os dois garfos simultaneamente. Portanto, não haverá impasse. Você pode tentar isso para N = 2, N = 3 e N = 4 e depois generalizar.

52. Um sistema de tempo real precisa tratar de duas chamadas de voz onde cada uma executa a cada 6 ms e consome 1 ms de tempo da CPU por surto, mais um vídeo de 25 quadros/s, com cada quadro exigindo 20 ms de tempo de CPU. Esse sistema é escalonável?
    a. 6/(25x20) = 0012 ms. O sistema é escalonável , pois consegue expandir sua capacidade de uso sem aumentar drasticamente custos com recursos técnicos ou capital humano.

53. Considere um sistema no qual se deseja separar política e mecanismo para o escalonamento de threads de núcleo. Proponha um meio de atingir essa meta.
    a. 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. 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 sãobloqueados após 5 ms, a ordem do thread por algum período de 30 ms pode ser A1, B1, A2, B2, A3, B3, algo que não é possível com esses parâmetros e threads de usuário. Essa situação está parcialmente descrita na Figura 2.44(b). Uma diferença importante entre threads de usuário e de núcleo é o desempenho. Realizar um chaveamento de thread com threads de usuário exige um punhado de instruções de máquina. Com threads de núcleo é necessário um chaveamento de contexto completo, mudar o mapa de memória e invalidar o cache, o que representa uma demora de magnitude várias ordens maior. Por outro lado, com threads de núcleo, ter um bloqueio de thread na E/S não suspende todo o processo como ocorre com threads de usuário. Visto que o núcleo sabe que chavear de um thread no processo A para um thread no processo B é mais caro do que executar um segundo thread no processo A (por ter de mudar o mapa de memória e invalidar a memória de cache), ele pode levar essa informação em conta quando toma uma decisão.

54. . Na solução para o problema do jantar dos filósofos (Figura 2.47), por que a variável de estado está configurada para HUNGRY na rotina take_forks?

a. Se um filósofo bloquear, os vizinhos poderão ver mais tarde que ela está com fome verificando seu estado, em teste, para que ele possa ser despertado quando os garfos estiverem disponíveis.

55. Considere a rotina put_forks na Figura 2.47. Suponha que a variável state[i] foi configurada para THINKING após as duas chamadas para teste, em vez de antes. Como essa mudança afetaria a solução?
    a. O programa usa um conjunto de semáforos, um por filósofo, portanto os filósofos com fome podem ser bloqueados se os garfos necessários estiverem ocupados. Observe que cada processo executa a rotina philosopher como seu código principal, mas as outras rotinas, take_forks, put_forks e test, são rotinas ordinárias e não processos separados.

56. O problema dos leitores e escritores pode ser formulado de várias maneiras em relação a qual categoria de processos pode ser iniciada e quando. Descreva cuidadosamente três variações diferentes do problema, cada uma favorecendo (ou não favorecendo) alguma categoria de processos. Para cada variação, especifique o que acontece quando um leitor ou um escritor está pronto para acessar o banco de dados, e o que acontece quando um processo foi concluído.
    a. Imagine, por exemplo, um sistema de reservas de uma companhia aérea, com muitos processos competindo entre si desejando ler e escrever. É aceitável ter múltiplos processos lendo o banco de dados ao mesmo tempo, mas se um processo está atualizando (escrevendo) o banco de dados, nenhum outro pode ter acesso, nem mesmo os leitores. A questão é: como programar leitores e escritores? Uma solução é mostrada na Figura 2.48. Nessa solução, para conseguir acesso ao banco de dados, o primeiro leitor realiza um down no semáforo db. Leitores subsequentes apenas incrementam um contador, rc. À medida que os leitores saem, eles decrementam o contador, e o último a deixar realiza um up no semáforo, permitindo que um escritor bloqueado, se houver, entre. A solução apresentada aqui contém implicitamente uma decisão sutil que vale observar. Suponha que enquanto um leitor está usando o banco de dados, aparece outro leitor. Visto que ter dois leitores ao mesmo tempo não é um problema, o segundo leitor é admitido. Leitores adicionais também podem ser admitidos se aparecerem. Agora suponha que um escritor apareça. O escritor pode não ser admitido ao banco de dados, já que escritores precisam ter acesso exclusivo, então ele é suspenso. Depois, leitores adicionais aparecem. Enquanto pelo menos um leitor ainda estiver ativo, leitores subsequentes serão admitidos. Como consequência dessa estratégia, enquanto houver uma oferta uniforme de leitores, todos eles entrarão assim que chegarem. O escritor será mantido suspenso até que nenhum leitor esteja presente. Se um novo leitor aparecer, digamos, a cada 2 s, e cada leitor levar 5 s para realizar o seu trabalho, o escritor jamais entrará. Para evitar essa situação, o programa poderia ser escrito de maneira ligeiramente diferente: quando um leitor chega e um escritor está esperando, o leitor é suspenso atrás do escritor em vez de ser admitido imediatamente. Dessa maneira, um escritor precisa esperar por leitores que estavam ativos quando ele chegou, mas não precisa esperar por leitores que chegaram depois dele. A desvantagem dessa solução é que ela alcança uma concorrência menor e assim tem um desempenho mais baixo. Courtois et al. (1971) apresentam uma solução que

dá prioridade aos escritores. Para detalhes, consulte o artigo.

57. Escreva um roteiro (script) shell que produz um arquivo de números sequenciais lendo o último número, adicionando 1 a ele, e então anexando-o ao arquivo. Execute uma instância do roteiro no segundo plano e uma no primeiro plano, cada uma acessando o mesmo arquivo. Quanto tempo leva até que a condição de corrida se manifeste? Qual é a região crítica? Modifique o roteiro para evitar a corrida. (Dica: use In file file.lock para travar o arquivo de dados.)

```
if [ ! –f numbers ]; then echo 0 > numbers; fi

count = 0

while (test $count != 200 )

do

        count=′expr $count + 1′

        n=′tail –1 numbers′

        expr $n + 1 >>numbers

done
```

a. Definição de uma trava no arquivo antes de entrar na área crítica e desbloqueie-a ao sair da área crítica.

```
if ln numbers numbers.lock

then

        n=′tail –1 numbers′

        expr $n + 1 >>numbers

        rm numbers.lock

fi
```

58. Presuma que você tem um sistema operacional que fornece semáforos. Implemente um sistema de mensagens. Escreva os procedimentos para enviar e receber mensagens.

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 */
     }
}
```

a.

59. Solucione o problema do jantar de filósofos usando monitores em vez de semáforos.

```
#define N 5                                     /* numero de filosofos */

void philosopher(int i)                         /* i: numero do filosofo, de 0 a 4 */
{
     while (TRUE) {
          think( );                             /* o filosofo esta pensando */
          take_fork(i);                         /* pega o garfo esquerdo */
          take_fork((i+1) % N);                 /* pega o garfo direito; % e o operador modulo */
          eat( );                               /* hummm, espaguete */
          put_fork(i);                          /* devolve o garfo esquerdo a mesa */
          put_fork((i+1) % N);                  /* devolve o garfo direito a mesa */
     }
}
```

a.