Questionário Disciplina Sistemas Operacionais

Prof Angel Antonio Gonzalez Martinez

Capítulo 02

Sumário

Sumário

[Capítulo 2 5](#_Toc146464780)

[Q01 (01 livro) 5](#_Toc146464781)

[Resposta: 5](#_Toc146464782)

[Q02 (02 livro) 5](#_Toc146464783)

[Resposta: 5](#_Toc146464784)

[Q03 (03 livro) 6](#_Toc146464785)

[Resposta: 6](#_Toc146464786)

[Q04 (04 livro) 6](#_Toc146464787)

[Resposta: 6](#_Toc146464788)

[Q05 (05 livro) 6](#_Toc146464789)

[Resposta: 6](#_Toc146464790)

[Q06 (06 livro) 7](#_Toc146464791)

[Resposta: 7](#_Toc146464792)

[Q07 (07 livro) 7](#_Toc146464793)

[Resposta: 7](#_Toc146464794)

[Q08 (08 livro) 7](#_Toc146464795)

[Resposta: 8](#_Toc146464796)

[Q09 (09 livro) 8](#_Toc146464797)

[Resposta: 8](#_Toc146464798)

[Q10 (10 livro) 8](#_Toc146464799)

[Resposta: 8](#_Toc146464800)

[Q11 (11 livro) 9](#_Toc146464801)

[Resposta: 9](#_Toc146464802)

[Q12 (12 livro) 9](#_Toc146464803)

[Resposta: 9](#_Toc146464804)

[Q13 (13 livro) 9](#_Toc146464805)

[Resposta: 9](#_Toc146464806)

[Q14 (14 livro) 10](#_Toc146464807)

[Resposta: 10](#_Toc146464808)

[Q15 (15 livro) 10](#_Toc146464809)

[Resposta: 10](#_Toc146464810)

[Q16 (16 livro) 10](#_Toc146464811)

[Resposta: 10](#_Toc146464812)

[Q17 (17 livro) 11](#_Toc146464813)

[Resposta: 11](#_Toc146464814)

[Q18 (18 livro) 11](#_Toc146464815)

[Resposta: 11](#_Toc146464816)

[Q19 (19 livro) 11](#_Toc146464817)

[Resposta: 12](#_Toc146464818)

[Q20 (20 livro) 12](#_Toc146464819)

[Resposta: 12](#_Toc146464820)

[Q21 (21 livro) 12](#_Toc146464821)

[Resposta: 12](#_Toc146464822)

[Q22 (22 livro) 13](#_Toc146464823)

[Resposta: 13](#_Toc146464824)

[Q23 (23 livro) 13](#_Toc146464825)

[Resposta: 13](#_Toc146464826)

[Q24 (24 livro) 13](#_Toc146464827)

[Resposta: 13](#_Toc146464828)

[Q25 (25 livro) 14](#_Toc146464829)

[Resposta: 14](#_Toc146464830)

[Q26 (26 livro) 14](#_Toc146464831)

[Resposta: 14](#_Toc146464832)

[Q27 (27 livro) 14](#_Toc146464833)

[Resposta: 14](#_Toc146464834)

[Q28 (28 livro) 15](#_Toc146464835)

[Resposta: 15](#_Toc146464836)

[Q29 (29 livro) 15](#_Toc146464837)

[Resposta: 15](#_Toc146464838)

[Q30 (31 livro) 15](#_Toc146464839)

[Resposta: 16](#_Toc146464840)

[Q31 (32 livro) 16](#_Toc146464841)

[Resposta: 16](#_Toc146464842)

[Q32 (33 livro) 16](#_Toc146464843)

[Resposta: 16](#_Toc146464844)

[Q33 (34 livro) 17](#_Toc146464845)

[Resposta: 17](#_Toc146464846)

[Q34 (36 livro) 17](#_Toc146464847)

[Resposta: 17](#_Toc146464848)

[Q35 (37 livro) 17](#_Toc146464849)

[Resposta: 17](#_Toc146464850)

# Capítulo 2

## Q01 (01 livro)

**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?

### Resposta:

A transição de bloqueado para executado é concebível. Suponha que um processo seja bloqueado na E/S e a E/S seja concluída. Se a CPU estiver ociosa, o processo pode ir diretamente de bloqueado para execução. A outra transição que falta, de pronta para bloqueada, é impossível. Um processo pronto não pode fazer E/S ou qualquer outra coisa que possa bloqueá-lo. Somente um processo em execução pode bloquear.

## Q02 (02 livro)

**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.

### Resposta:

Você pode ter um registro contendo um ponteiro para a entrada da tabela de processo atual. Quando a E/S for concluída, a CPU armazenará o estado atual da máquina na entrada atual da tabela de processos. Em seguida, ele iria para o vetor de interrupção para o dispositivo de interrupção e buscaria um ponteiro para outra entrada de tabela de processo (o procedimento ser-vice). Esse processo seria então iniciado.

## Q03 (03 livro)

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

### Resposta:

Geralmente, as linguagens de alto nível não permitem o tipo de acesso ao hardware da CPU que é necessário. Por exemplo, um manipulador de interrupção pode ser necessário para habilitar e desabilitar a manutenção de interrupção de um dispositivo específico ou para manipular dados dentro da área de pilha de um processo. Além disso, as rotinas de serviço de interrupção devem ser executadas o mais rápido possível.

## Q04 (04 livro)

**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?

### Resposta:

Existem várias razões para usar uma pilha separada para o kernel. Dois deles são os seguintes. Primeiro, você não quer que o sistema operacional trave porque um programa de usuário mal escrito não permite espaço suficiente na pilha. Em segundo lugar, se o kernel deixar dados de pilha no espaço de memória de um programa de usuário ao retornar de uma chamada do sistema, um usuário mal-intencionado poderá usar esses dados para descobrir informações sobre outros processos.

## Q05 (05 livro)

**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?

### Resposta:

A chance de que todos os cinco processos estejam ociosos é de 1/32, portanto, o tempo ocioso da CPU é de 1/32.

## Q06 (06 livro)

**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?

### Resposta:

Há espaço suficiente para 14 processos na memória. Se um processo tem uma E/S de p, então a probabilidade de que todos eles estejam aguardando E/S é p14. Igualando-se a 0,01, obtém-se a equação p14 = 0. 01. Resolvendo isso, obtemos p=0.72 , para que possamos tolerar processos com até 72% de espera de E/S.

## Q07 (07 livro)

**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%.

### Resposta:

Se cada trabalho tiver 50% de espera de E/S, levará 40 minutos para ser concluído na ausência de competição. Se executado sequencialmente, o segundo terminará 80 minutos após o início do primeiro. Com dois trabalhos, a utilização aproximada da CPU é 1 – (0.5)2. Assim, cada um recebe 0,375 minuto de CPU por minuto de tempo real. Para acumular 20 minutos de tempo de CPU, um trabalho deve ser executado por 20/0,375 minutos, ou cerca de 53,33 minutos. Assim, correndo sequencialmente, os trabalhos terminam após 80 minutos, mas correndo em paralelo terminam após 53,33 minutos.

## Q08 (08 livro)

**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?

### Resposta:

A probabilidade de que todos os processos estejam aguardando E/S é de 0,46, que é 0,004096. Portanto, a utilização da CPU = 1  0. 004096  0: 995904.

## Q09 (09 livro)

**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.

### Resposta:

O processo do cliente pode criar threads separados; Cada thread pode buscar uma parte diferente do arquivo de um dos servidores espelho. Isso pode ajudar a reduzir o tempo de inatividade. Claro, há um único link de rede sendo compartilhado por todos os threads. Esse link pode se tornar um gargalo à medida que o número de threads se torna muito grande.

## Q10 (10 livro)

**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?

### Resposta:

Seria difícil, se não impossível, manter o sistema de arquivos consistente. Suponha que um processo de cliente envie uma solicitação ao processo 1 do servidor para atualizar um arquivo. Esse processo atualiza a entrada de cache em sua memória. Logo em seguida, outro processo do cliente envia uma solicitação ao servidor 2 para ler esse arquivo. Infelizmente, se o arquivo também estiver armazenado em cache lá, o servidor 2, em sua inocência, retornará dados obsoletos. Se o primeiro processo gravar o arquivo no disco depois de armazená-lo em cache, e o servidor 2 verificar o disco em cada leitura para ver se sua cópia armazenada em cache está atualizada, o sistema pode ser feito para funcionar, mas são precisamente todos esses acessos ao disco que o sistema de cache está tentando evitar.

## Q11 (11 livro)

**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?

### Resposta:

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

## Q12 (12 livro)

**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?

### Resposta:

Um thread de trabalho será bloqueado quando precisar ler uma página da Web do disco. Se threads de nível de usuário estiverem sendo usados, essa ação bloqueará todo o processo, destruindo o valor do multithreading. Assim, é essencial que os threads do kernel sejam usados para permitir que alguns threads bloqueiem sem afetar os outros.

## Q13 (13 livro)

**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.

### Resposta:

Sim. Se o servidor estiver totalmente vinculado à CPU, não há necessidade de ter vários threads. Isso apenas adiciona complexidade desnecessária. Como exemplo, considere um número de assistência de lista telefônica (como 555-1212) para uma área com 1 milhão de pessoas. Se cada registro (nome, número de telefone) for, digamos, 64 caracteres, o banco de dados en- tire leva 64 megabytes e pode ser facilmente mantido na memória do servidor para fornecer pesquisa rápida.

## Q14 (14 livro)

**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.

### Resposta:

Quando um thread é interrompido, ele tem valores nos registros. Eles devem ser salvos, assim como quando o processo é interrompido. Os registros devem ser salvos. Threads multiprogram- ming não é diferente de processos de multiprogramação, então cada thread precisa de sua própria área de salvamento de registro.

## Q15 (15 livro)

**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.

### Resposta:

Threads em um processo cooperam. Eles não são hostis uns aos outros. Se o rendimento for necessário para o bem da aplicação, então um thread renderá. Afinal, geralmente é o mesmo programador que escreve o código para todos eles.

## Q16 (16 livro)

**16.** É possível que um thread seja antecipado por uma interrupção de relógio? Se a resposta for afirmativa, em quais circunstâncias?

### Resposta:

Os threads no nível do usuário não podem ser antecipados pelo relógio, a menos que todo o quantum do processo tenha sido usado (embora interrupções de relógio transparentes possam acontecer). Os threads no nível do kernel podem ser antecipados individualmente. Neste último caso, se um thread for executado por muito tempo, o relógio interromperá o processo atual e, portanto, o thread atual. O kernel é livre para escolher um thread diferente do mesmo processo para executar em seguida, se assim desejar.

## Q17 (17 livro)

**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?

### Resposta:

No caso de thread único, os acertos de cache levam 12 mseg e os erros de cache levam 87 mseg. A média ponderada é de 2/3 x 12 + 1/3 x 87. Assim, a requisição média leva 37 mseg e o servidor pode fazer cerca de 27 por segundo. Para um servidor multithread, toda a espera pelo disco é sobreposta, de modo que cada solicitação leva 12 ms, e o servidor pode lidar com 83 solicitações 1/3 por segundo.

## Q18 (18 livro)

**18.** Qual é a maior vantagem de se implementar threads no espaço de usuário? Qual é a maior desvantagem?

### Resposta:

A maior vantagem é a eficiência. Não são necessárias armadilhas para o kernel para alternar threads. A maior desvantagem é que, se um thread bloqueia, todo o processo bloqueia.

## Q19 (19 livro)

**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?

### Resposta:

Sim, isso pode ser feito. Após cada chamada para pthread create, o programa principal pode fazer uma junção pthread para aguardar até que o thread recém criado tenha saído antes de criar o próximo thread.

## Q20 (20 livro)

**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?

### Resposta:

Os ponteiros são realmente necessários porque o tamanho da variável global é desconhecido. Pode ser qualquer coisa, desde um caractere até uma matriz de números de ponto flutuante. Se o valor fosse armazenado, teríamos que dar o tamanho para criar global, o que é bom, mas que tipo deve ser o segundo parâmetro de set global, e que tipo deve ser o valor de read global?

## Q21 (21 livro)

**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?

### Resposta:

Pode acontecer que o sistema de tempo de execução esteja precisamente no ponto de bloquear ou desbloquear um thread e esteja ocupado manipulando as filas de agendamento. Este seria um momento muito inoportuno para o manipulador de interrupção de relógio começar a inspecionar essas filas para ver se era hora de fazer a troca de threads, já que elas podem estar em um estado inconsistente. Uma solução é definir um sinalizador quando o sistema de tempo de execução é inserido. O manipulador do relógio veria isso e definiria sua própria bandeira, depois retornaria. Quando o sistema de tempo de execução terminasse, ele verificaria o sinalizador de relógio, veria que ocorreu uma interrupção de relógio e agora executaria o manipulador de relógio.

## Q22 (22 livro)

**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.

### Resposta:

Sim, é possível, mas ineficiente. Um thread que deseja fazer uma chamada do sistema primeiro define um temporizador de alarme e, em seguida, faz a chamada. Se a chamada for bloqueada, o temporizador retornará o controle para o pacote de threads. É claro que, na maioria das vezes, a chamada não será bloqueada, e o temporizador precisa ser limpo. Assim, cada chamada de sistema que pode bloquear tem que ser executada como três chamadas de sistema. Se os temporizadores disparam prematuramente, todos os tipos de problemas se desenvolvem. Esta não é uma maneira atraente de criar um pacote de threads.

## Q23 (23 livro)

**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?

### Resposta:

Sim, ainda funciona, mas ainda está ocupado esperando, é claro

## Q24 (24 livro)

**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?

### Resposta:

Certamente funciona com agendamento preventivo. Na verdade, ele foi projetado para esse caso. Quando o agendamento não é preventivo, ele pode falhar. Considere o caso em que o turno é inicialmente 0, mas o processo 1 é executado primeiro. Ele simplesmente fará um loop para sempre e nunca liberará a CPU.

## Q25 (25 livro)

**25.** O problema da inversão de prioridades discutido na Seção 2.3.4 acontece com threads de usuário? Por que ou por que não?

### Resposta:

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. Com threads de nível de usuário, não pode acontecer que um thread de baixa prioridade seja subitamente antecipado para permitir um execução de thread de alta prioridade. Não há preempção. Com threads no nível do kernel esse problema pode surgir.

## Q26 (26 livro)

**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.

### Resposta:

Com o agendamento de ida e volta, funciona. Mais cedo ou mais tarde L correrá e, eventualmente, deixará sua região crítica. A questão é que, com o agendamento prioritário, L nunca consegue correr, com round robin, ele recebe uma fatia de tempo normal periodicamente, então tem a chance de deixar sua região crítica.

## Q27 (27 livro)

**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.

### Resposta:

Cada thread chama procedimentos por conta própria, portanto, ele deve ter sua própria pilha para as variáveis locais, endereços de retorno e assim por diante. Isso é igualmente verdadeiro para threads no nível do usuário como para threads no nível do kernel.

## Q28 (28 livro)

**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?

### Resposta:

Sim. O computador simulado pode ser multiprogramado. Por exemplo, enquanto o processo A está em execução, ele lê alguma variável compartilhada. Em seguida, um tique-taque 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 ele também adiciona 1 à variável, temos uma condição de corrida.

## Q29 (29 livro)

**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?

### Resposta:

Sim, vai funcionar como está. Em um determinado instante de tempo, apenas um produtor (consumidor) pode adicionar (remover) um item ao buffer.

## Q30 (31 livro)

**31.** Como um sistema operacional capaz de desabilitar interrupções poderia implementar semáforos?

### Resposta:

Para fazer uma operação de semáforo, o sistema operacional primeiro desabilita as interrupções. Em seguida, lê-se o valor do semáforo. Se ele está fazendo um down e o semáforo é igual a zero, ele coloca 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 forem bloqueados, um deles será removido da lista de processos bloqueados e tornado executável. Quando todas essas operações tiverem sido concluídas, as interrupções poderão ser ativadas novamente

## Q31 (32 livro)

**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.

### Resposta:

Associados a cada semáforo de contagem estão dois semáforos binários, M, usado para exclusão mútua, e B, usado para bloqueio. Também associado a cada semáforo de contagem está um contador que contém o número de subidas menos o número de baixas, e uma lista de processos bloqueados nesse semáforo. Para implementar, um processo primeiro ganha acesso exclusivo aos semáforos, contador e lista fazendo um down em M. Em seguida, decrementa o balcão. Se for zero ou mais, ele só faz um up em M e sai. Se M for negativo, o processo é colocado na lista de processos bloqueados. Em seguida, um up é feito em M e um down é feito em B para bloquear o processo. Para implementar, primeiro M é derrubado para obter mútuo e, em seguida, o contador é incrementado. Se for mais de zero, ninguém foi bloqueado, então tudo o que precisa ser feito é subir M. Se, no entanto, o contador agora for negativo ou zero, algum processo deve ser removido da lista. Finalmente, um up é feito em B e M nessa ordem.

## Q32 (33 livro)

**33.** Se um sistema tem apenas dois processos, faz sentido usar uma barreira para sincronizá-los? Por que ou por que não?

### Resposta:

Se o programa opera em fases e nenhum processo pode entrar na próxima fase até que ambos estejam terminados com a fase atual, faz todo o sentido usar uma barreira.

## Q33 (34 livro)

**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.

### Resposta:

Com threads do kernel, um thread pode bloquear em um semáforo e o kernel pode executar algum outro thread no mesmo processo. Consequentemente, não há problema em usar semáforos. Com threads em nível de usuário, quando um thread bloqueia em um sem- áforo, o kernel acha que todo o processo está bloqueado e não o executa nunca mais. Consequentemente, o processo falha

## Q34 (36 livro)

**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.

### Resposta:

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

## Q35 (37 livro)

**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?

### Resposta:

Não leva a condições de corrida (nada é perdido), mas está efetivamente ocupado esperando.