UNIVERSIDADE FEDERAL DO PIAUÍ - UFPI

CURSO: BACHAREL EM SISTEMAS DE INFORMAÇÃO

DISCIPLINA: SISTEMAS OPERACIONAIS

PROFESSOR: DEBORAH MAGALHÃES

Aluno: MAURICIO BENJAMIN DA ROCHA

LISTA DE EXERCÍCIO II

1. **Qual a motivação por trás do conceito de threads?**

   A principal motivação é o desempenho obtido ao usar threads.

   Uma thread pode ser considerada como uma unidade básica de utilização da CPU dentro de um processo. Ela compartilha o mesmo espaço de endereçamento e recursos do processo pai, mas possui sua própria pilha de execução. Dessa forma, várias threads podem executar em paralelo dentro do mesmo processo, permitindo que várias tarefas sejam realizadas simultaneamente.

2. **Explique a diferença entre um ambiente multithread e monothread?**
   - Execução Simultânea: No ambiente multithread, várias threads podem ser executadas ao mesmo tempo, possibilitando a realização de tarefas em paralelo, enquanto no ambiente monothread, apenas uma thread é executada por vez, resultando em execução sequencial de tarefas.
   - Paralelismo: O ambiente multithread pode aproveitar o paralelismo disponível em sistemas multi-core ou multi-processadores, distribuindo as threads para diferentes núcleos de CPU, melhorando o desempenho geral. Já o ambiente monothread não pode aproveitar plenamente essas configurações de hardware para executar tarefas simultaneamente.
   - Complexidade: O uso de várias threads em um ambiente multithread pode aumentar a complexidade do código, exigindo sincronização adequada para evitar condições de corrida e outros problemas de concorrência. No ambiente monothread, a complexidade é menor, pois a execução é estritamente sequencial.
   - Eficiência: O ambiente multithread pode ser mais eficiente em termos de recursos, pois cria threads mais leves em comparação com a criação de novos processos no ambiente monothread, o que consome mais recursos.

Em resumo, o ambiente multithread oferece maior potencial para melhorar o desempenho e a responsividade de programas, especialmente em sistemas com vários núcleos de CPU ou tarefas paralelizáveis. Já o ambiente monothread é mais simples e adequado para tarefas que não exigem paralelismo ou não se beneficiariam significativamente de uma abordagem multithread.

3. **Por que o gerenciamento de threads é mais eficiente que o de processos?**
    - Custo de Recursos: Criar e destruir processos é uma operação relativamente custosa em termos de recursos do sistema. Isso ocorre porque cada processo tem seu próprio espaço de endereçamento, tabela de descritores de arquivos e outras estruturas de dados associadas. Em contraste, as threads compartilham o mesmo espaço de endereçamento e outras estruturas de dados comuns dentro de um processo, o que reduz significativamente o custo computacional em comparação com a criação de um novo processo.
    - Comunicação e Sincronização: A comunicação e a sincronização entre threads são mais eficientes do que entre processos. Como as threads compartilham o mesmo espaço de endereçamento, a comunicação entre elas pode ser realizada através de variáveis compartilhadas na memória, sem a necessidade de mecanismos complexos de comunicação entre processos, como pipes ou filas de mensagens. Além disso, a sincronização entre threads pode ser feita com primitivas de sincronização, como semáforos ou mutexes, que são mais leves em termos de recursos em comparação com os mecanismos de sincronização entre processos.
    - Trocas de Contexto Rápidas: Mudar de uma thread para outra dentro do mesmo processo envolve uma troca de contexto mais rápida do que mudar de um processo para outro. Isso ocorre porque, como as threads compartilham o mesmo espaço de endereçamento, o sistema operacional precisa atualizar apenas algumas estruturas de dados para realizar a troca de contexto, ao contrário de uma troca de processo, que requer a atualização de todo o espaço de endereçamento.
    - Paralelismo Eficiente: Em sistemas com vários núcleos de CPU, as threads podem ser atribuídas a diferentes núcleos, permitindo que sejam executadas em paralelo real, aproveitando o potencial de processamento paralelo do hardware. Isso pode levar a um melhor desempenho geral do sistema em comparação com a execução sequencial de processos separados.

- Resposta do Sistema: Devido à sua natureza mais leve e ao compartilhamento eficiente de recursos, as threads permitem uma resposta mais rápida do sistema a eventos concorrentes. Isso é particularmente importante em aplicativos interativos ou com interface do usuário, onde a responsividade é essencial.

4. **Quais informações são compartilhadas entre as threads de um processo? Quais informações são específicas de cada thread?**

   Informações Compartilhadas entre as Threads:

   - Espaço de Endereçamento: Todas as threads dentro de um mesmo processo compartilham o mesmo espaço de endereçamento. Isso significa que elas podem acessar as mesmas variáveis globais e dados compartilhados na memória do processo.
   - Descritores de Arquivos e Tabelas: As threads compartilham os mesmos descritores de arquivos e tabelas de arquivos abertos do processo. Isso permite que todas as threads acessem os mesmos arquivos abertos e recursos de E/S.
   - Código Executável: Todas as threads de um processo compartilham o mesmo código executável. Isso significa que, em vez de carregar várias cópias do programa na memória, apenas uma cópia do código é carregada e compartilhada entre todas as threads.
   - Sinais e Tratadores de Sinais: As threads compartilham os tratadores de sinais do processo. Quando um sinal é recebido pelo processo, qualquer thread pode tratá-lo, pois os sinais são entregues a todo o processo, não a uma thread específica.

   Informações Específicas de Cada Thread:

   - Registradores e Contadores: Cada thread possui seus próprios registradores e contadores de programa (como o Program Counter - PC). Isso permite que cada thread mantenha seu estado de execução independente das outras threads.
   - Pilha de Execução: Cada thread tem sua própria pilha de execução para armazenar os registros de ativação, parâmetros de função e outras informações necessárias durante a execução de funções. A pilha é individual para cada thread e é usada para manter o rastro da execução de chamadas de função e retornos.
   - Variáveis Locais: As variáveis locais em cada função são específicas de cada thread. Cada thread tem sua própria cópia das variáveis locais em suas funções, permitindo que cada thread tenha um estado independente durante a execução.

- Estado do Processo: Embora as threads compartilhem o mesmo espaço de endereçamento, cada thread possui um estado de execução independente. Isso significa que, em um determinado momento, diferentes threads podem estar em diferentes pontos de execução no mesmo código, executando diferentes partes do programa.

5. **Em que consiste uma thread?**

   Uma thread, também conhecida como "subprocesso" ou "encadeamento", é uma unidade básica de execução dentro de um processo. Ela representa uma sequência de instruções que podem ser executadas de forma independente em um sistema operacional. Em outras palavras, uma thread é uma linha de execução dentro de um programa, com seu próprio contador de programa (Program Counter - PC), pilha de execução e estado de registradores.

6. **O que é um pacote de threads e quais são seus diferentes modos?**

   Um pacote de threads, também conhecido como biblioteca de threads ou modelo de programação de threads, é um conjunto de funções e rotinas que permitem aos desenvolvedores criar, gerenciar e sincronizar threads em seus programas. Essas bibliotecas facilitam a implementação e o uso eficiente de threads, fornecendo uma interface de programação de aplicativos (API) para trabalhar com threads de forma mais conveniente e abstrata.

   Os diferentes modos de um pacote de threads geralmente se referem à forma como as threads são gerenciadas e agendadas pelo sistema operacional. Existem dois modos principais de implementação de threads:

- Threads em Espaço de Usuário (User-level Threads - ULTs): Nesse modo, o gerenciamento de threads é realizado em nível de aplicativo, sem o envolvimento direto do sistema operacional. A biblioteca de threads no espaço do usuário gerencia todas as operações relacionadas a threads, como criação, escalonamento, sincronização e destruição. O sistema operacional só é ciente do processo como um todo, não percebendo as threads individuais. Isso pode resultar em uma troca de contexto mais rápida, mas também limita a capacidade de tirar proveito do paralelismo de hardware em sistemas multi-core, já que o escalonamento é feito em nível de usuário.
- Threads em Espaço de Kernel (Kernel-level Threads - KLTs): Nesse modo, as threads são gerenciadas pelo próprio sistema operacional, e cada thread é tratada como uma entidade independente pelo escalonador do kernel. O sistema operacional tem conhecimento detalhado de cada thread e pode escaloná-las em diferentes núcleos

de CPU, aproveitando o paralelismo de hardware de forma mais eficiente. No entanto, a troca de contexto entre threads em espaço de kernel pode ser mais lenta do que em ULTs, já que envolve uma interação com o kernel.

Além desses dois modos principais, existem variações e combinações, como a utilização de "Threads em Espaço de Usuário sobre Kernel" (User-level Threads over Kernel Threads - UoLTK), que tenta combinar as vantagens dos dois métodos, permitindo ao programador trabalhar com threads em espaço de usuário e utilizar threads em espaço de kernel para aproveitar o paralelismo de hardware.

7. **Quais as vantagens e desvantagens de implementar threads em modo kernel?**

Vantagens:

- Paralelismo Eficiente: Com KLTs, o escalonador do kernel pode atribuir threads individuais para diferentes núcleos de CPU, permitindo que as threads sejam executadas em paralelo real e aproveitem o paralelismo de hardware de forma mais eficiente. Isso pode levar a um melhor desempenho geral do sistema em sistemas multi-core ou multi-processadores.
- Sincronização Mais Segura: Ao utilizar o escalonador do kernel, a sincronização entre threads pode ser mais segura, pois o kernel tem uma visão mais completa e precisa do estado de cada thread. Isso pode reduzir a possibilidade de problemas de concorrência, como condições de corrida.
- Prioridades de Escalonamento: O escalonador do kernel pode usar prioridades para definir a importância relativa de diferentes threads. Isso permite que threads de alta prioridade sejam executadas antes de threads de baixa prioridade, o que pode ser útil para dar mais recursos às tarefas mais importantes.

Desvantagens:

- Overhead de Troca de Contexto: A troca de contexto entre threads em espaço de kernel geralmente é mais lenta do que em ULTs. Isso ocorre porque a mudança para o modo kernel e o envolvimento do sistema operacional adicionam alguma sobrecarga ao processo de troca de contexto.
- Escalabilidade Limitada: Embora o paralelismo de hardware possa ser mais bem explorado, a implementação em modo kernel pode limitar a escalabilidade, especialmente em cenários com um grande número de threads concorrentes. O escalonador do kernel precisa lidar com todas as threads do sistema, o que pode se tornar mais complexo e menos eficiente à medida que o número de threads aumenta.

- Custo de Recursos: O uso de threads em modo kernel pode consumir mais recursos do sistema, pois cada thread é tratada como uma entidade independente pelo sistema operacional. Isso pode resultar em maior consumo de memória e CPU.
- Dependência do Sistema Operacional: A implementação de threads em modo kernel está mais vinculada ao sistema operacional específico, tornando o código mais dependente do SO. Isso pode dificultar a portabilidade do código entre diferentes sistemas operacionais.

Em resumo, a implementação de threads em modo kernel oferece um melhor aproveitamento do paralelismo de hardware e uma sincronização mais segura, mas pode ser menos eficiente em termos de troca de contexto e pode ter uma escalabilidade limitada em comparação com ULTs.

8. **Quais as vantagens e desvantagens de implementar threads em modo usuário?**

Vantagens:

- Troca de Contexto Mais Rápida: Em ULTs, a troca de contexto entre threads é geralmente mais rápida, pois é realizada em nível de aplicativo, sem envolver o sistema operacional diretamente. Isso ocorre porque as trocas de contexto são feitas apenas entre as threads dentro do espaço de usuário, envolvendo menos operações e overhead em comparação com KLTs.
- Escalabilidade Potencialmente Melhorada: Como a troca de contexto é mais rápida em ULTs, pode haver um potencial de escalabilidade melhorado em cenários com muitas threads concorrentes, especialmente em sistemas que têm uma carga de trabalho intensiva em termos de criação e destruição frequente de threads.
- Portabilidade: ULTs são geralmente mais portáveis entre diferentes sistemas operacionais, pois dependem menos das características e funcionalidades específicas do kernel do SO. Isso torna o código mais independente do SO e facilita a portabilidade para ambientes diferentes.
- Maior Controle: A implementação em modo usuário permite mais controle sobre o escalonamento e a sincronização das threads, já que é possível personalizar e implementar algoritmos específicos no nível da aplicação.

Desvantagens:

- Sincronização Mais Complexa: Em ULTs, a sincronização entre threads pode ser mais complexa, pois não há uma visão completa do estado de todas as threads por parte do sistema operacional. Isso pode levar a problemas de concorrência, como condições de corrida, que precisam ser tratados de forma cuidadosa e precisa pela aplicação.

- Limitações no Paralelismo de Hardware: ULTs não aproveitam plenamente o paralelismo de hardware em sistemas multi-core ou multi-processadores, pois o escalonamento das threads é feito em nível de usuário, sem a capacidade de atribuir threads a núcleos de CPU específicos.
- Bloqueio de Threads: Se uma thread realizar uma operação bloqueante, como entrada/saída (I/O), ela pode bloquear todo o processo, afetando outras threads que também executam dentro do mesmo espaço de usuário.
- Dependência do Pacote de Threads: ULTs normalmente dependem de bibliotecas de threads específicas que fornecem as funções de gerenciamento de threads. A escolha do pacote de threads correto é importante para garantir o funcionamento adequado das ULTs.

Em resumo, a implementação de threads em modo usuário oferece uma troca de contexto mais rápida e potencialmente uma melhor escalabilidade, além de ser mais portável entre diferentes sistemas operacionais. No entanto, a sincronização pode ser mais complexa e pode haver limitações no aproveitamento total do paralelismo de hardware.

9. **O conceito a seguir, se refere a qual das alternativas abaixo: 2 ou mais processos estão lendo ou escrevendo em uma região de memória compartilhada ao mesmo tempo.**

   ( ) Exclusão mútua
   ( ) Região crítica
   ( ) Comunicação entre processos
   (X) Condição de corrida

10. **Qual das opções abaixo não corresponde a uma boa solução de exclusão mútua:**

    ( ) Nenhuma suposição pode ser feita a respeito de velocidades ou do número de CPUs

    ( ) Nenhum processo deve ser obrigado a esperar eternamente para entrar em sua região crítica

    (X) Nenhum processo deve ser impedido de entrar em sua região crítica

    ( ) Nenhum processo executando fora de sua região crítica pode bloquear qualquer processo

11. **Quais das alternativas abaixo corresponde ao conceito de preempção :**

    () o processo é executado sem tempo determinado até acabar

    ( ) o processo executado é aquele que possui menor tempo de execução

(X) o processo é executado por um intervalo de tempo pré-definido

( ) o processo executado é aquele que chegou primeiro

12. Defina o que é exclusão mútua com espera ocupada.

A exclusão mútua com espera ocupada é um mecanismo de sincronização usado para garantir que apenas uma thread por vez tenha acesso a uma seção crítica do código. Nesse método, uma thread que deseja entrar na seção crítica fica em um loop ativo (espera ocupada) até que possa obter o acesso exclusivo.

Em resumo, a exclusão mútua com espera ocupada é uma abordagem onde uma thread espera de forma ativa até que possa entrar em uma região crítica, evitando assim que outras threads acessem a mesma região simultaneamente.

13. Explique como a solução de Peterson garante a exclusão mútua.

A solução de Peterson utiliza duas variáveis de turno e uma variável booleana para cada thread para garantir que apenas uma thread por vez entre na região crítica. Vamos supor que temos duas threads, T0 e T1, e cada uma delas tem suas próprias variáveis booleanas para representar o interesse em entrar na região crítica e uma variável turno para indicar qual thread tem prioridade.

O algoritmo funciona da seguinte forma:

1. Fase de Interesse: Cada thread, antes de entrar na região crítica, define sua variável booleana correspondente como True, indicando que está interessada em entrar.

2. Definição do Turno: A variável turno é definida para a thread que não está interessada em entrar na região crítica, ou seja, para a outra thread. Isso dá prioridade à outra thread para acessar a região crítica.

3. Espera Ocupada: Cada thread entra em um loop de espera ativa (espera ocupada), onde verifica repetidamente se a outra thread está interessada em entrar na região crítica e se é a vez dela (se turno == self). Se ambas as condições forem verdadeiras, significa que a outra thread tem prioridade, então a thread atual aguarda até que seja a sua vez e a outra thread não esteja mais interessada em entrar na região crítica.

4. Acesso à Região Crítica: Quando é a vez da thread atual (turno == self) e a outra thread não está interessada em entrar na região crítica, a thread atual pode entrar na região crítica e executar suas operações protegidas.

5. Saída da Região Crítica: Após concluir as operações na região crítica, a thread redefine sua variável booleana para False, indicando que não está mais interessada em entrar, permitindo que a outra thread possa ter a chance de entrar na região crítica.

A solução de Peterson garante a exclusão mútua, pois as threads esperam ativamente pela vez da outra e não entram em conflito na região crítica. Dessa forma, apenas uma thread pode acessar a região crítica por vez, garantindo que não ocorram condições de corrida ou acesso simultâneo a recursos compartilhados. No entanto, essa solução é específica para apenas duas threads e pode se tornar mais complexa ao lidar com mais threads.

14. Explique o problema do Produtor/Consumidor.

O problema do Produtor/Consumidor é um clássico problema de concorrência que envolve a comunicação e a sincronização entre duas entidades: um produtor, que produz dados, e um consumidor, que consome esses dados. Ambos os processos estão sendo executados simultaneamente e compartilham um buffer (ou fila) com tamanho limitado.

O objetivo é garantir que o produtor não insira dados no buffer quando estiver cheio e que o consumidor não retire dados do buffer quando estiver vazio. Além disso, é necessário evitar condições de corrida ou bloqueios indevidos durante a inserção e a retirada de dados.

O problema pode ser resumido da seguinte maneira:

1. O produtor produz dados e tenta inseri-los no buffer. Se o buffer estiver cheio, o produtor deve esperar até que haja espaço disponível para inserir os dados.

2. O consumidor retira dados do buffer e consome-os. Se o buffer estiver vazio, o consumidor deve esperar até que haja dados disponíveis para consumir.

3. O buffer tem uma capacidade limitada para armazenar os dados produzidos pelo produtor. Ele deve ser gerenciado de forma apropriada para garantir que não haja acesso simultâneo inadequado, o que pode resultar em dados perdidos ou corrompidos.

Para resolver o problema do Produtor/Consumidor, são utilizados mecanismos de sincronização, como semáforos, mutexes ou outras primitivas de sincronização, para controlar o acesso ao buffer. A sincronização adequada entre o produtor e o consumidor é essencial para evitar problemas de concorrência, como condições de corrida, deadlock ou inanição.

Existem várias abordagens para resolver o problema do Produtor/Consumidor, incluindo soluções baseadas em semáforos, monitores, filas bloqueantes, etc. Cada abordagem tem suas próprias características e complexidades, e a escolha depende do contexto específico do sistema e das necessidades do programa.

O problema do Produtor/Consumidor é um exemplo clássico de como a concorrência pode introduzir desafios significativos no projeto de sistemas e como a sincronização correta é fundamental para garantir a correta execução e a consistência dos dados em ambientes multi-threaded ou multi-processadores.

15. Qual a diferença entre as abordagens de escalonamento de sistemas interativos e sistemas em lote, incluindo seus objetivos específicos?

As abordagens de escalonamento de sistemas interativos e sistemas em lote são diferentes devido aos objetivos específicos de cada um e às características das tarefas que são executadas em cada tipo de sistema.

Escalonamento de Sistemas Interativos:

1. Objetivo: O principal objetivo do escalonamento em sistemas interativos é fornecer uma resposta rápida ao usuário. Esses sistemas geralmente são utilizados para executar aplicativos que têm interação direta com o usuário, como interfaces gráficas, sistemas de gerenciamento de janelas e aplicativos de escritório.

2. Prioridade à Responsividade: Nesse tipo de escalonamento, a prioridade é dada ao atendimento rápido das solicitações do usuário. É importante garantir que o sistema responda prontamente a entradas do teclado, cliques do mouse e outras ações do usuário, proporcionando uma experiência interativa fluida.

3. Tempo Compartilhado: Sistemas interativos geralmente usam o conceito de tempo compartilhado, no qual cada processo ou thread recebe uma pequena fatia de tempo da CPU antes de ser interrompido para dar chance a outros processos em execução. Isso permite que várias tarefas concorrentes sejam executadas rapidamente em pequenos intervalos de tempo.

4. Sistemas Multiusuários: Sistemas interativos costumam ser multiusuários, permitindo que vários usuários interajam com o sistema simultaneamente, e o escalonamento é projetado para garantir que cada usuário tenha uma resposta rápida e justa do sistema.

Escalonamento de Sistemas em Lote:

1. Objetivo: O objetivo principal do escalonamento em sistemas em lote é maximizar a eficiência do processamento de grandes volumes de trabalhos que normalmente requerem processamento em lote sem intervenção direta do usuário. Esses sistemas são comuns em ambientes de processamento de dados, computação científica e processamento em lote de grandes tarefas.

2. Prioridade ao Throughput: Nesse tipo de escalonamento, a prioridade é dada ao throughput (quantidade de trabalho processado por unidade de tempo), em vez da resposta rápida ao usuário. O foco está na execução eficiente de trabalhos em lote sem a necessidade de interação contínua com os usuários.

3. Tempos de Execução Mais Longos: Em sistemas em lote, é comum que os trabalhos tenham tempos de execução mais longos, o que exige um planejamento adequado para otimizar o uso da CPU e garantir que todos os trabalhos sejam concluídos no menor tempo possível.

4. Escalonamento por Prioridade e Critérios de Prazo: Em sistemas em lote, o escalonador pode utilizar critérios de prioridade baseados em fatores como prazo (deadline) de execução ou prioridade de cada trabalho, a fim de garantir que trabalhos críticos ou urgentes sejam processados a tempo.

Em resumo, a principal diferença entre as abordagens de escalonamento de sistemas interativos e sistemas em lote está na priorização de objetivos: sistemas interativos priorizam a resposta rápida ao usuário e o tempo compartilhado, enquanto sistemas em lote priorizam o throughput e a eficiência do processamento de grandes volumes de trabalhos em lote. As características específicas dos processos e o contexto do sistema determinam qual abordagem de escalonamento é mais adequada.

16. Explique a solução para o problema produtor/consumidor baseado em semáforos.

A solução do problema do Produtor/Consumidor baseada em semáforos envolve o uso de dois semáforos e um buffer (ou fila) compartilhado entre o produtor e o consumidor. Os semáforos são usados para sincronizar e controlar o acesso ao buffer, garantindo que o produtor não insira dados quando o buffer estiver cheio e que o consumidor não retire dados quando o buffer estiver vazio.

Basicamente:

1. Semáforos: Utilizam-se dois semáforos:

- Semáforo cheio: Inicializado com 0, indica a quantidade de espaços ocupados no buffer.
- Semáforo vazio: Inicializado com o tamanho total do buffer, indica a quantidade de espaços disponíveis no buffer.

2. Buffer Compartilhado: É uma área de memória compartilhada entre o produtor e o consumidor, onde os dados produzidos pelo produtor são armazenados e consumidos pelo consumidor.

3. Produtor: Quando o produtor deseja inserir um dado no buffer, ele primeiro adquire o semáforo vazio (se houver espaço disponível no buffer). Em seguida, insere o dado no buffer e libera o semáforo cheio, indicando que há um espaço ocupado no buffer.

4. Consumidor: Quando o consumidor deseja retirar um dado do buffer, ele primeiro adquire o semáforo cheio (se houver dados no buffer). Em seguida, retira o dado do buffer e libera o semáforo vazio, indicando que há um espaço disponível no buffer para inserção de novos dados.

17. Quais as primitivas de troca de mensagem e quais os desafios dessa abordagem?

As primitivas de troca de mensagem são as operações de inserção (produção) e remoção (consumo) de itens no buffer compartilhado pelos produtores e consumidores. Já os desafios são: exclusão mútua, sincronização, deadlock, tamanho do Buffer, coordenação entre produtores e consumidores.

18. Qual a diferença entre um algoritmo de escalonamento preemptivo e não preemptivo?

No escalonamento preemptivo, o sistema operacional pode interromper um processo em execução e passá-lo para o estado de pronto, com o objetivo de alocar outro processo na UCP. No escalonamento não-preemptivo, quando um processo está em execução, nenhum evento externo pode ocasionar a perda do uso do processador.

19. Por que a preempção é fundamental para sistemas interativos?

Porque esses sistemas têm como característica principal a necessidade de fornecer uma resposta rápida e interativa aos usuários. E permite que o sistema operacional gerencie de forma eficaz a execução de tarefas, priorizando aquelas que são mais importantes para a interação com o usuário e respondendo prontamente a eventos externos.

20. Quais os objetivos gerais de um algoritmo de escalonamento?

Eles têm por objetivo realizar o ordenamento temporal de um conjunto de tarefas, gerenciando a atualização da fila de execução, dinamicamente ou não, dependendo da técnica utilizada. Alguns dos objetivos do algoritmo de escalonamento incluem minimizar o tempo de resposta em sistemas interativos, principalmente servidores, e decidir qual dos processos prontos para execução deve ser alocado à CPU.

21. Explique o funcionamento dos seguintes algoritmos de escalonamento:

(i)Escalonamento Circular (Round-Robin);

(ii) Escalonamento por prioridades;

(iii) Escalonamento por loteria;

(i) Escalonamento Circular; (Round-Robin)

É um algoritmo de escalonamento preemptivo que consiste em repartir uniformemente o tempo da CPU entre todos os processos prontos para a execução. Os processos são organizados numa fila circular, alocando-se a cada um uma fatia de tempo da CPU, igual a um número inteiro de quanta. Caso um processo não termine dentro de sua fatia de tempo, ele é colocado no fim da fila e uma nova fatia de tempo é alocada para o processo no começo da fila 1.

(ii) Escalonamento por prioridades

É um algoritmo de escalonamento que atribui prioridades aos processos e os executa com base nessas prioridades. Processos com prioridades mais altas são executados antes dos processos com prioridades mais baixas. As prioridades podem ser estáticas (definidas pelo sistema ou pelo usuário) ou dinâmicas (alteradas pelo sistema durante a execução).

(iii) Escalonamento por loteria

É um algoritmo de escalonamento que atribui “bilhetes de loteria” aos processos e seleciona aleatoriamente um “bilhete” vencedor para determinar qual processo será executado a seguir. Processos com mais bilhetes têm uma chance maior de serem selecionados. Os bilhetes podem ser atribuídos com base em vários critérios, como prioridade ou uso de recursos.

22. O que diferencia o escalonamento por fração justa dos descritos na questão anterior.

O Escalonamento por Fração Justa (Fair-Share) é um algoritmo de escalonamento que garante que cada usuário tenha sua fração da CPU, independentemente do número de processos que cada usuário tenha. A fração alocada ao usuário é garantida pelo escalonador conforme a “noção” de justiça sobre a prioridade do usuário. Isso difere dos outros algoritmos mencionados anteriormente, pois eles se concentram em atribuir tempo de CPU aos processos individuais, enquanto o Escalonamento por Fração Justa se concentra em garantir que cada usuário tenha uma fração justa do tempo de CPU.