Universidade Federal de Lavras
PPGCC
PCC508 – Sistemas Operacionais

Lista Avaliativa 3

Douglas Aquino T. Mendes
28 de outubro de 2024

Sumário

1 Perguntas

1.1 O que é uma condição de corrida

Pergunta: 1) Explique o que é uma condição de corrida o como a exclusão mútua de regiões críticas pode ser utilizada para prevenir o problema

Resposta: Uma condição de corrida é um problema que ocorre em sistemas computacionais quando dois ou mais processos ou threads acessam e manipulam dados compartilhados de forma concorrente, levando a resultados indesejados ou inconsistentes. Esse problema surge em ambientes onde a ordem de execução das operações não é controlada, resultando em situações em que o comportamento do sistema depende da sequência de execução dos processos.

A exclusão mútua é uma técnica de controle de concorrência que garante que apenas um processo ou thread possa acessar uma região crítica de código por vez. Uma região crítica é qualquer parte do código que acessa recursos compartilhados, como variáveis ou estruturas de dados, e onde a manipulação incorreta pode levar a inconsistências ou condições de corrida [1], [2].

1.2 Obter exclusão mútua

Pergunta: 2) Como pode-se obter exclusão mútua utilizando-se a TSL? E desabilitando interrupções? São soluções boas? Quais as desvantagens?

Resposta: A exclusão mútua pode ser implementada utilizando a instrução TSL (Test and Set Lock), que é uma operação atômica utilizada para garantir que apenas um processo ou thread acesse uma região crítica de código por vez. A instrução TSL realiza duas operações atômicas, Testa o valor de um lock (bloqueio) e Define o valor do lock para ocupado, se o lock estiver livre. Se o lock já estiver ocupado, a instrução simplesmente retorna o valor atual do lock.

A implementação com TSL geralmente resulta em um spinlock, onde uma thread ou processo fica em loop ativo (busy-waiting) até que consiga adquirir o lock. Isso pode ser ineficiente, especialmente em sistemas com alta contenda, pois consome ciclos de CPU.

A exclusão mútua também pode ser implementada desabilitando interrupções,essa abordagem garante que um processo ou thread possa acessar uma região crítica sem ser interrompido, evitando condições de corrida.

A desabilitação de interrupções pode ser eficiente em sistemas de tempo real ou em sistemas embarcados, onde é crítico garantir que uma operação não seja interrompida. No entanto, em sistemas multitarefa mais complexos, essa técnica pode levar a problemas de desempenho.

1.3 Exclusão mútua utilizando Mutexes

Pergunta: 3) Explique como funciona o método de exclusão mútua utilizando Mutexes, abordando sua implementação na biblioteca Pthreads e também variáveis de condição

Resposta: O método de exclusão mútua utilizando mutexes é uma abordagem utilizada em programação concorrente para garantir que apenas um thread acesse uma região crítica de código por vez. A implementação de mutexes e variáveis de condição na biblioteca Pthreads (POSIX Threads) é uma forma de gerenciar a sincronização entre threads em sistemas POSIX, como Linux e UNIX.

Um mutex é um objeto de sincronização que permite que somente um thread bloqueie e acesse uma seção crítica do código. O mutex possui dois estados, bloqueado (locked) e livre (unlocked). Quando um thread tenta adquirir um mutex, se o mutex estiver livre, o thread o bloqueia e pode entrar na região crítica. Se o mutex já estiver bloqueado por outro thread, o thread atual será colocado em espera até que o mutex seja liberado.

A biblioteca Pthreads fornece funções para criar e gerenciar mutexes. Abaixo estão as principais etapas para implementar um mutex em Pthreads [1], [3]:

Passo 1: Incluir o Cabeçalho Pthread

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

Passo 2: Declarar e Inicializar o Mutex

```
pthread_mutex_t mutex;  // Declaração do mutex

// Inicialização do mutex
pthread_mutex_init(&mutex, NULL);
```

Passo 3: Usar o Mutex na Região Crítica

```
void *thread_function(void *arg) {
    pthread_mutex_lock(&mutex);  // Tenta adquirir o mutex

    // Região Crítica: código que acessa recursos compartilhados

    pthread_mutex_unlock(&mutex);  // Libera o mutex
    return NULL;
}
```

Passo 4: Finalizar o Mutex Após o uso, o mutex deve ser destruído:

```
pthread_mutex_destroy(&mutex);  // Destrói o mutex
```

As variáveis de condição são utilizadas em conjunto com mutexes para permitir que threads esperem por certas condições antes de prosseguir com a execução. Elas são úteis em situações onde um thread deve esperar por uma condição ser verdadeira antes de acessar uma região crítica ou um recurso compartilhado.

Implementação de Variáveis de Condição em Pthreads

Passo 1: Declarar e Inicializar a Variável de Condição

```
pthread_cond_t cond_var;  // Declaração da variável de condição
pthread_mutex_t mutex;    // Declaração do mutex

// Inicialização
pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond_var, NULL);
```

Passo 2: Usar a Variável de Condição

```
void *producer(void *arg) {
    pthread_mutex_lock(&mutex);

    // Produz um item e modifica uma condição
    // Se necessário, sinaliza a condição para que outro thread possa prosseguir
    pthread_cond_signal(&cond_var);

    pthread_mutex_unlock(&mutex);
    return NULL;
}

void *consumer(void *arg) {
    pthread_mutex_lock(&mutex);
```

```
    // Espera até que a condição seja verdadeira
    while (/* condição não satisfeita */) {
        pthread_cond_wait(&cond_var, &mutex);  // Espera e libera o mutex
    }

    // Consome um item

    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

Passo 3: Finalizar a Variável de Condição

```
pthread_cond_destroy(&cond_var);  // Destrói a variável de condição
pthread_mutex_destroy(&mutex);     // Destrói o mutex
```

1.4 Round Robin

Pergunta: Descreva, com exemplo, como funciona o escalonamento conhecido como "Round Robin".

Resposta: O escalonamento Round Robin (RR) é um algoritmo de gerenciamento de processos baseado na atribuição de um tempo fixo de CPU a cada processo em uma fila de prontos, permitindo que todos os processos tenham uma oportunidade equitativa de se executar.

Funcionamento do Round Robin

1. **Fila de Prontos**: Os processos que estão prontos para serem executados são organizados em uma fila.
2. **Tempo de Quantum**: Cada processo é atribuído a um tempo fixo de CPU, denominado *quantum*. Quando um processo utiliza o quantum, ele é colocado no final da fila e o próximo processo da fila é iniciado.
3. **Preempção**: Se um processo não terminar sua execução dentro do tempo de quantum, ele é interrompido (preempted), e o controle é passado para o próximo processo na fila.
4. **Retorno à Fila**: O processo que foi interrompido retorna ao final da fila de prontos, aguardando sua vez de ser escalonado novamente.
5. **Ciclo**: O ciclo se repete até que todos os processos sejam concluídos.

Exemplo Considere um cenário com três processos: P_1, P_2, P_3, como ilustrado na tabela 1, e um tempo de quantum de 4 unidades de tempo.

Processo	Tempo de Chegada	Tempo de Execução
P_1	0	8
P_2	1	4
P_3	2	9

Tabela 1: Processos com Tempo de Chegada e Execução

Escalonamento com Round Robin

1. **Execução do Processo** P_1: O primeiro processo na fila é P_1. Ele é executado por 4 unidades de tempo (de 0 a 4), restando 4 unidades.
2. **Execução do Processo** P_2: O próximo na fila é P_2. Ele é executado por 4 unidades de tempo (de 4 a 8), e agora é concluído (0 unidades restantes).
3. **Execução do Processo** P_3: O próximo na fila é P_3. Ele é executado por 4 unidades de tempo (de 8 a 12), restando 5 unidades.
4. **Retorno do Processo** P_1: O algoritmo volta para P_1, que foi interrompido anteriormente. Ele é executado por 4 unidades (de 12 a 16) e agora está concluído.
5. **Retorno do Processo** P_3: O próximo na fila é P_3. Ele é executado por 4 unidades (de 16 a 20), restando 1 unidade.
6. **Finalização do Processo** P_3: O algoritmo volta para P_3 novamente, e ele é executado por 1 unidade (de 20 a 21), agora completando sua execução.

Tabela de Execução A tabela 2 resume a execução:

Tempo	Executando
0-4	P_1
4-8	P_2
8-12	P_3
12-16	P_1
16-20	P_3
20-21	P_3

Tabela 2: Tabela de Execução do Algoritmo Round Robin

1.5 Prioridades dinâmicas

Pergunta: O algoritmo de escalonamento baseado em prioridades dinâmicas, onde a prioridade é calculada com 1/f (f = fração do quantum utilizada), prioriza qual tipo de processo? Qual a vantagem de priorizar esse tipo de processo?

Resposta: O algoritmo de escalonamento baseado em prioridades dinâmicas prioriza processos que utilizam menos tempo de CPU em relação ao quantum alocado. Em outras palavras, quanto menor a fração do quantum que um processo utiliza, maior será sua prioridade. Processos interativos se beneficiam dessa abordagem pois são processos que frequentemente requerem resposta do usuário e tendem a usar menos tempo de CPU por interação, pois aguardam entradas do usuário ou eventos externos.

Ao priorizar processos que consomem menos tempo de CPU, o sistema melhora a responsividade geral onde atrasos podem resultar em uma experiência do usuário insatisfatória. E esse tipo de escalonamento ajuda a evitar a monopolização da CPU por processos que consomem muito tempo.

2 Desenvolver um programa

2.1 06) Solução de Peterson.

Enunciado: Nesse exercício, um contador compartilhado será acessado por 2 threads simultaneamente. Elas devem pegar o valor do contador, imprimir na tela, executar threadyield, somar um no contador. Duas versões devem ser feitas: uma sem nenhum controle de condição de corrida e outra utilizando a solução de Peterson para isso

Rsposta: Os codigos estão nos arquivos atv6_s.c e atv6_pat.c, como base foi utilizado a solução do livro SISTEMAS OPERACIONAIS MODERNOS [1], págs.: 85 e 86.

Referências

[1] A. S. Tanenbaum e H. Bos, *SISTEMAS OPERACIONAIS MODERNOS*, 4ª ed. Boston: Pearson, 2021.

[2] Embarcados, "Condição de Corrida em Sistemas Embarcados," 2023, Acesso em: 28 out. 2024. endereço: `https://embarcados.com.br/condicao-de-corrida-em-sistemas-embarcados/`.

[3] J. Nona, *UML e suas Aplicações*, Acesso em: 28 out. 2024, 2013. endereço: `https://gist.github.com/jonatasnona/5371859`.