FACULDADE DE ENGENHARIA

DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA

Engenharia Informática

Sistemas Operativos e Programação Concorrente

Grupo 1 – Programação Concorrente

Docente: Eng°. Délcio Chadreca

Discentes:

Cassamo, Kayron Lourenço

Firme, Alfoi Renato

Mandlate, Enoque Admiro

Mungoi, Belmiro

Xavier, Omar Davide

Maputo, Abril de 2023

1.Índice

2. Resumo ..... **Erro! Marcador não definido.**

3. Palavras-chave: 4

4. Introdução 3

5. Objectivos 4

6. Metodologia ..... **Erro! Marcador não definido.**

7. Programação Concorrente 6

7.1. Programas Multithreads 6

7.1.1. Definição de threads 6

7.1.2. Modelo de Processo & Modelo de Threads 7

7.1.3. Estados de threads, ciclo de vida de uma thread 7

7.1.4. Tipos de Threads 8

7.2. Comunicação e sincronização de processos 9

7.3. Primitivas de sincronização 16

7.3.1.1. Semáforos 16

7.3.1.2. Mutexes 17

7.3.1.3. Monitores 18

7.3.1.4. Barreiras de sincronização 19

7.3.2. Implementação e utilização de primitivas de sincronização em diferentes linguagens de programação. 20

7.4. Problemas clássicos de programação concorrente 21

7.4.1. Produtores/consumidores 21

7.4.2. Jantar dos Filósofos 23

7.5. Construções concorrentes de alto nível 25

8. Conclusão 267

9. Referências bibliográficas 278

2. Resumo

A programação concorrente é uma técnica de programação que permite que dois ou mais processos ou threads executem simultaneamente em um sistema de computador. Isso aumenta a eficiência do processamento, pois vários processos podem ser executados ao mesmo tempo, em vez de um processo ter que esperar o término de outro processo para começar a executar.

Os programas concorrentes podem ser implementados usando várias técnicas, incluindo threads, processos, comunicação e sincronização. Threads são sequências independentes de instruções que podem ser executadas simultaneamente em um único processo. Processos são programas independentes que podem ser executados simultaneamente em um sistema operacional. A comunicação e sincronização são necessárias para garantir que os processos ou threads cooperem entre si e evitem conflitos.

As primitivas de sincronização, como semáforos, mutexes e monitores, são usadas para controlar o acesso a recursos compartilhados entre processos ou threads, garantindo que apenas um processo ou thread possa acessá-lo em um determinado momento. Além disso, elas garantem que as operações que dependem da ordem de execução sejam realizadas na ordem correta.

Os problemas clássicos da programação concorrente incluem o problema do produtor/consumidor, o problema do leitor/escritor e o problema do jantar dos filósofos, que ilustram as dificuldades envolvidas na coordenação de processos ou threads que compartilham recursos.

Construções concorrentes de alto nível são abstrações da programação que podem ajudar a simplificar a implementação de programas concorrentes com o uso de threads e reduzir os erros de sincronização.

3. Palavras-chave:

- Concorrência
- Threads
- Processos
- Programas
- Sincronização
- Recurso

4. Introdução

A programação concorrente é uma abordagem na qual múltiplos fluxos de execução (threads) trabalham simultaneamente em um mesmo programa. Essa técnica pode melhorar significativamente o desempenho e a eficiência de um software, pois permite a execução paralela de diversas tarefas, aproveitando melhor os recursos disponíveis no sistema. No entanto, a programação concorrente também traz desafios, especialmente em relação à sincronização e comunicação entre as threads, de modo a garantir a consistência dos dados e evitar problemas como condições de corrida e deadlocks. Neste trabalho, serão apresentados os principais conceitos sobre programas multithreads, a comunicação entre processos que é necessária para que os mesmos possam cooperar entre si na execução de tarefas, as primitivas de sincronização, que são mecanismos utilizados para implementar a sincronização em programas concorrentes, os problemas clássicos em programação concorrente e por fim as construções concorrentes de alto nível.

5. Objectivos

Objectivo geral

- Apresentar uma análise detalhada da programação concorrente, descrevendo seu funcionamento e importância na computação moderna.

Objectivos específicos

- Descrever os conceitos fundamentais da programação concorrente, incluindo processos, threads, sincronização, e comunicação entre processos.
- Discutir os principais desafios da programação concorrente.
- Apresentar exemplos de aplicação da programação concorrente em sistemas do mundo real.

6. Metodologia

Para atingir os objetivos deste relatório, foi realizada uma revisão detalhada da literatura disponível sobre programação concorrente. Foram consultados artigos acadêmicos, livros, sites especializados e outros recursos confiáveis que apresentam informações relevantes sobre o assunto. Estes recusrsos estão nas referências bibliográficas.

7. Programação Concorrente

A programação concorrente é uma técnica de programação que permite que dois ou mais processos ou threads executem simultaneamente em um sistema de computador. Isso aumenta a eficiência do processamento, pois vários processos podem ser executados ao mesmo tempo, em vez de um processo ter que esperar o término de outro processo para começar a executar.

7.1. Programas Multithreads

7.1.1. Definição de threads

Uma thread (linha de execução) é uma unidade basica de processamento que pode ser executada simultaneamente com outras threads dentro de um processo.

As threads permitem que um programa realize multiplas tarefas simultaneamente, melhorando o desempenho e a eficiência geral do sistema. Elas são amplamente utilizadas em aplicativos que precisam lidar com tarefas assincronas, como aplicativos de rede, processamento de dados, interface gráfica de usuário e jogos.

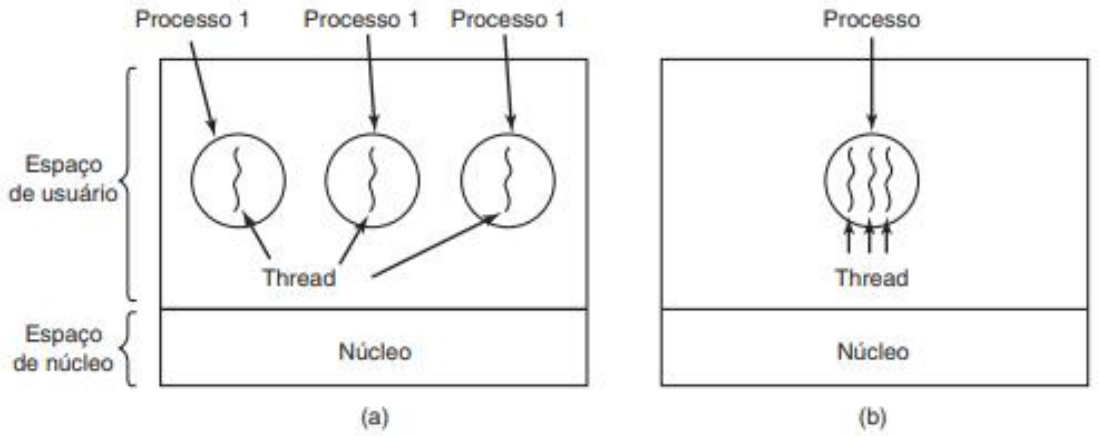


Figura 2-6 (a) Três processos, cada um com uma *thread*. (b) Um processo com três *threads*.

A idéia de multithreading é associar vários fluxos de execução (várias threads) a um único processo.

7.1.2. Modelo de Processo & Modelo de Threads

Embora uma thread deva ser executada em algum processo, a thread e seu processo são conceitos diferentes e podem ser tratados separadamente. Os processos são usados para agrupar recursos;

Itens por processo	**Itens por threads**
• Espaço de endereçamento • Variavis globais • Arquivos abertos • Processos filhos • Alarmes pendentes • Sinais e rotinas de tratamento de sinais • Informações de contabilização	• Contador de programa • Registradores • Pilha de execução • Estado

Figura:2-7 A primeira coluna lista alguns itens compartilhados por todas as threads em um processo. A segunda lista alguns itens privativos de cada threaad.

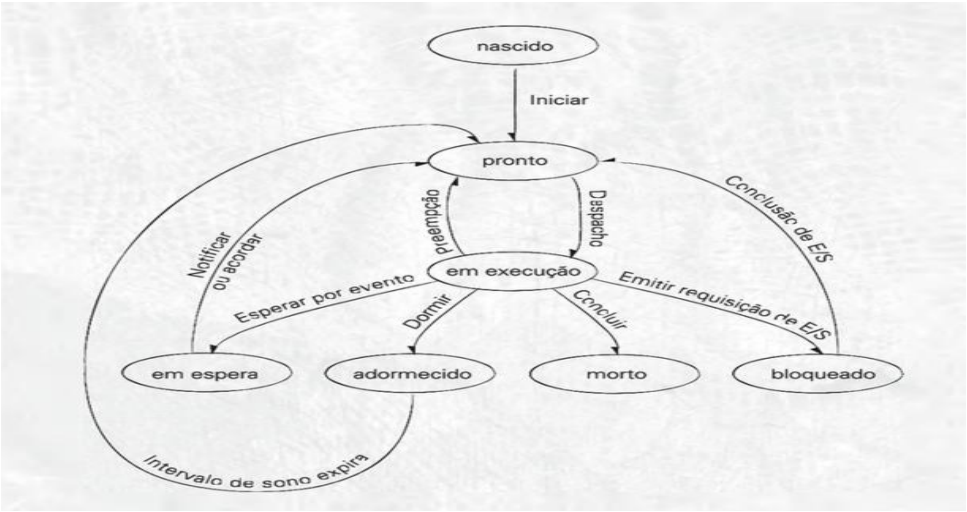
O que as threads acrescentam no modelo de processo é o fato de permitir que várias execuções ocorram no mesmo ambiente de processo de forma bastante independente umas da outras.

7.1.3. Estados de threads, ciclo de vida de uma thread

Uma thread pode estar em um dos seguintes estados de execução:

1- Novo (nascido): Quando uma thread é criada, ela está no estado novo. Nesse estado, o sistema operacional está configurando os recursos necessarios para que a thread seja executada.
2- Pronta (Ready): Esperando para ser executada.

3- Executando (Running): sendo executada pela CPU.
4- Bloqueado: aguardando por algum recurso, como entrada/saida.
5- Espera: espera por um evento.
6- Suspensa/adormecido (Suspended): temporariamente interrompida e pode ser retomada posteriormente.
7- Morto (Dead): conclui sua execucao e não pode mais ser executada.



7.1.4. Tipos de Threads

Existem dois tipos de threads:

1- **Thread de usuário**: são criadas e gerenciadas pelo proprio processo do usuario, sem a intervencao do kernel. Essas threads são leves e são executadas em espaco de usuario, sem a necessidade de mudar para o proprio kernel. Elas são mais flexiveis do que as threads do sistema e permitem que o aplicativo tenha mais controle sobre o gerenciamento de threads, incluindo o escalonamento e o controle de concorrencia. No entanto, o sistema operacional não esta ciente das threads de usuario e, portanto, não pode escalona-las de forma eficiente entre os nucleos de processamento.

2- **Thread de sistema (thread do kernel)** : são criadas e gerenciadas pelo kernel do sistema operacional. Elas são mais pesadas do que as threads de usuario e exigem uma mudança para o modo kernel para serem executadas.

Programas multithreads

São programas que usam multiplas threads para executar várias tarefas simultaneamente em um mesmo processo. Os sistemas operacionais modernos permitem que os programas sejam executados de forma concorrente, usando a tecnologia de threads.

Os programas multithreads oferecem diversas vantagens, como melhor aproveitamento dos recursos de processamento e a possibilidade de manter a interface grafica do usuario responsivo, enquanto o programa executa tarefas em segundo plano.

Porem, a programação multithread requer cuidados especiais com relação a sincronização do acesso aos recursos compartilhados e prevencao de condiçoes de corrida, que podem levar a erros de execução e falhas no programa. Alem disso, programas multithread podem ser mais dificeis de depurar e testar devido a complexidade adicional introduzida pela concorrencia

As threads podem ser gerenciadas pelo sistema operacional ou pela propria aplicacao, dependendo de linguagem de programacao e do ambiente em que estao sendo usadas. É importante ter cuidado ao trabalhar com Threads, ja que problemas como condicoes de corridda e bloqueio podem ocorrer se elas nao forem gerenciadas correctamente.

As Threads podem ser gerenciadas pelo sistema operacional ou pela própria aplicação, dependendo da linguagem de programação e do ambiente em questão sendo usadas.

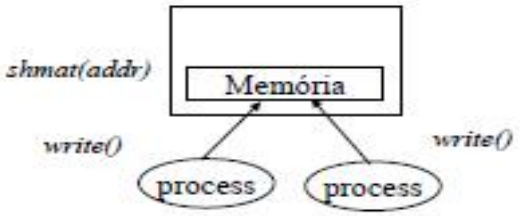
7.2. Comunicação e sincronização de processos

Em uma aplicação concorrente, ou na própria execução do sistema, muitas vezes é necessário que processos ou *threads* troquem dados entre si.

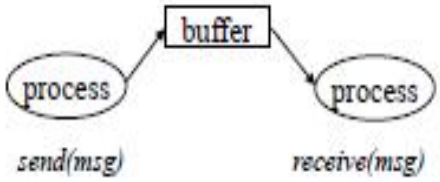
Tal comunicação pode ser implementada de diversas formas, por exemplo, usando um arquivo compartilhado, memória compartilhada, ou por troca de mensagens. Nesses casos, é necessário que os processos tenham suas execuções sincronizadas.

Comunicação e sincronização entre processos: duas abordagens.

1. **Baseada em memória compartilhada**
   - Comunicação e implícita (dados compartilhados, sem canal de comunicação);
   - Sincronização para acesso precisa ser feita explicitamente.



2. **Baseada em troca de mensagens:**
   - Canal de comunicação e explícito;
   - Sincronização e implícita (processos bloqueiam nas primitivas)



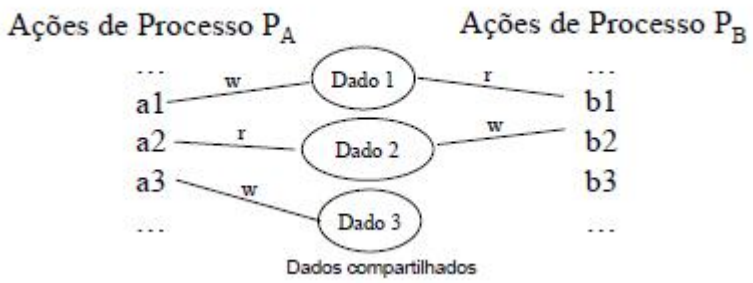
Condição de Corrida

Ocorre sempre que:

• Cada processo/*thread* precisa executar uma sequência de ações (a1,..aN) que envolvem mais de um dado/recurso compartilhado;

• Os dados/recursos precisam manter um estado consistente entre si;

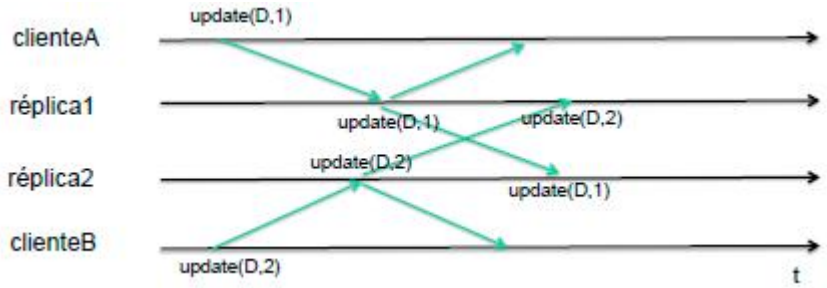
• Antes que complete a execução de toda a sequência de ações, um dos processos é interrompido pelo outro.



Também vale para troca de mensagens (entre processos clientes e processos servidores)

Exemplo:

1. Suponha um serviço tolerante a falhas que seja implementado com replicação (distribuída) de dados.
2. Assim que recebe uma requisição, um servidor responde ao cliente e envia a atualização seu dados para o outro servidor, que atualiza seu dado correspondentemente.



Os servidores precisam entrar em um consenso sobre a ordem final das operações.

Para evitar os problemas de condição de corrida, os processos/*theads* que compartilham algum dado/recurso:

1. Precisam “bloquear” temporariamente o recurso, impedindo que os outros processos interfiram no seu acesso ao recurso. (ou seja, garantir a Exclusão mútua)

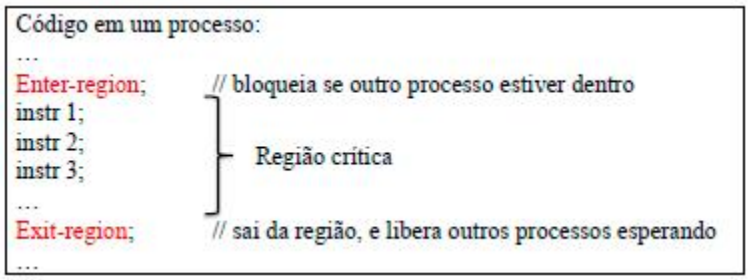
2. Precisam ser capazes de “notificar” os demais processos quando o recurso foi “liberado” ou quando está no estado que permita uma determinada operação
(por exemplo, consumo de dados quando um buffer deixou de estar vazio)
Esse “bloquear”, “liberar” e “notificar” precisa ser atômico.
Os locais no código em que há acesso a dados/recursos compartilhados precisa estar protegido.

Região Critica

Região crítica (ou Sessão crítica) é uma parte do código (programa) que contém as operações sobre dados compartilhados, a serem executadas em exclusão mútua.

Para implementar regiões críticas alguns requisitos básicos precisam ser satisfeitos:
Exclusão Mútua: Se um processo P está executando sua região crítica nenhum outro poderá executar a sua região crítica (segurança = safety).
Progresso: Um processo executando dentro da região crítica não pode ser bloqueado/terminado por outro processo (progresso = liveness).
Espera Limitada: Um processo não deve esperar indefinidamente para entrar em sua região crítica (justiça);

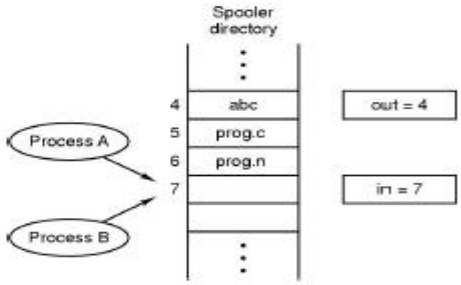
Para implementar uma região crítica deve haver um mecanismo/protocolo para garantir a entrada e saída segura (sincronizada, coordenada) desta parte do código.



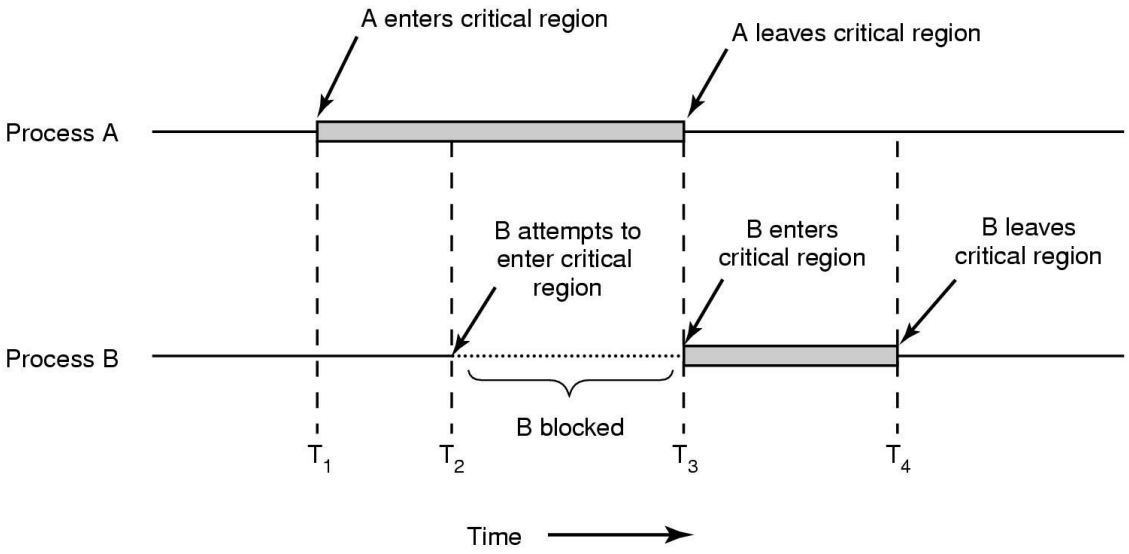
Exemplo de utilização de RCs:

```
Process {
…
        Enter-Region()
        copy file to Spooler[in];
        in = (in+1)%slots;
        Exit-Region()
…
}

Spooler {
   while(1) {
        Enter-Region();
        print Spooler[out];
        out = (out+1)%slots;
        Exit-Region();
   }
}
```



Região Critica



Exclusão mútua usando Regiões Críticas

Possíveis formas de Implementar Regiões Criticas

Alternativas:

1. **Desabilitar interrupções**:

   - Pode ser feito pelo núcleo, mais não por um processo em modo usuário.

2. **Processos compartilham uma flag "lock":**

   - Se lock=0, trocar valor para 1 e processo entra RC, senão processo espera;

   - Se leitura e atribuição do lock não for atómica, então exclusão mutua não e garantida.

3. **Espera Ocupada**:

   - Alternância regular de acesso por dois processos (PID= 0; PID= 1);

   - Só funciona para 2 processos e apenas se a frequência de acesso ao recurso for mais ou menos a mesma.

```
while (TRUE) {                          while (TRUE) {
    while (turn != 0)       /* loop */ ;     while (turn != 1)       /* loop */ ;
    critical_region( );                     critical_region( );
    turn = 1;                               turn = 0;
    noncritical_region( );                  noncritical_region( );
}                                       }
```

Região Critica com Espera Ocupada

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 - process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Solução de Peterson:

- Variável turn e o vector interested[ ] são variáveis compartilhadas;
- Se dois processos PID= {0,1} executam simultaneamente enter_region, o primeiro valor de turn será sobrescrito (e o processo correspondente vai entrar), mais interested[first] vai manter o registro do interesse do segundo processo;

Exclusão Mútua com Espera Ocupada

Algumas arquiteturas possuem uma instrução de máquina especial:

Test- and- Set- Lock (TSL):

Algumas arquiteturas possuem uma instrução de máquina TSL para leitura de um lock para um registrador e atribuição de um valor diferente de zero ($\neq 0$) de forma atómica!

Processos que desejam entrar RC executam TSL:

- Se registrador (e lock= 0), então entram na RC, senão esperam em loop

```
enter_region:
    TSL REGISTER,LOCK          | copy lock to register and set lock to 1
    CMP REGISTER,#0            | was lock zero?
    JNE enter_region           | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0               | store a 0 in lock
    RET | return to caller
```

Espera Ocupada vs. Bloqueio

- Solução de Peterson e Test- and – Set- Lock apresentam o problema da espera ocupada que consome ciclos de processamento.
- Outro problema da espera Ocupada: Inversão de prioridades
  Se um processo com baixa prioridade estiver na RC, demora mais a ser escalonado (e sair da RC), pois processos de maior prioridade estarão executando em espera ocupada (para a RC).

A alternativa:

Chamadas ao núcleo que bloqueiam o processo e o fazem esperar por um sinal de outro processo.

Exemplo:

- Lock/ unlock (Mutex);
- Wait/Signal;
- Semáforos.

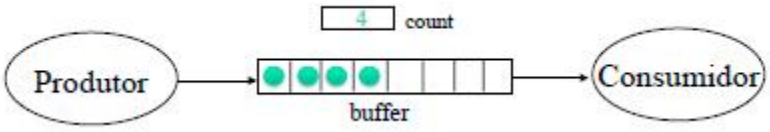
Problema do Produtor e Consumidor

Sincronização de dois processos que compartilham um buffer (um produz itens, o outro consome itens do buffer), e que usam uma variável compartilhada count para controlar o fluxo de controle.

- Se count=N, produtor deve esperar;

- Se count=0 consumidor deve esperar.

Cada processo deve “acordar” o outro processo quando o conteúdo do buffer mudar a ponto de permitir o prosseguimento do processamento.



Esse tipo de sincronização esta relacionada ao estado do recurso -> Sincronização de condição.

7.3. Primitivas de sincronização

A sincronização é a maneira de garantir que as tarefas sejam executadas de maneira organizada e previsível, evitando problemas como condições de corrida, em que duas ou mais tarefas tentam acessar o mesmo recurso ao mesmo tempo.

Na programação concorrente, a sincronização pode ser alcançada por meio de várias técnicas, como semáforos, monitores, mutexes e barreiras de sincronização. Essas técnicas ajudam a garantir que as tarefas sejam executadas de maneira ordenada, permitindo que elas compartilhem recursos de maneira segura e eficiente.

Por exemplo, se duas threads precisam acessar um arquivo simultaneamente, a sincronização garante que apenas uma thread tenha acesso ao arquivo de cada vez. Isso evita que os threads escrevam no arquivo ao mesmo tempo e acabem sobrescrevendo informações importantes. A sincronização também pode ser usada para garantir que uma thread espere outra thread antes de prosseguir, permitindo a execução em paralelo de várias tarefas sem problemas de dependência.

7.3.1.1. Semáforos

Os semáforos são uma das primitivas de sincronização mais utilizadas em programação concorrente. Eles consistem em uma variável inteira que é usada para controlar o acesso a um recurso compartilhado entre as threads ou processos. Os semáforos são úteis para evitar problemas de condição de corrida, deadlock e starvation em programação concorrente, pois eles

permitem que as threads ou processos sincronizem seu acesso a recursos compartilhados de maneira segura e controlada.

Existem dois tipos de semáforos: binários e contadores. O semáforo binário pode ter apenas dois valores possíveis: 0 e 1. Ele é usado para indicar se um recurso está disponível ou não. Por exemplo, em um programa que possui uma seção crítica, o semáforo binário pode ser usado para indicar se a seção crítica está sendo usada ou não. Já o semáforo contador é um inteiro que pode ter valores maiores que 1. Ele é usado para controlar o acesso a um recurso compartilhado que pode ser utilizado por várias threads ou processos simultaneamente, mas que possui um limite máximo de uso. Os semáforos possuem duas operações básicas: wait() e signal(). A operação wait() é usada para solicitar o acesso ao recurso compartilhado, e a operação signal() é usada para liberar o acesso ao recurso compartilhado.

Por exemplo, suponha que duas threads queiram acessar um recurso compartilhado e um semáforo com valor inicial 1 é usado para coordenar o acesso. Quando a primeira thread verifica o valor do semáforo, ele é igual a 1, então a primeira thread pode acessar o recurso e decrementa o valor do semáforo para 0. Quando a segunda thread verifica o valor do semáforo, ele é igual a 0, então a segunda thread chama wait() para esperar e é bloqueada até que a primeira thread libere o recurso e sinalize o semáforo através da operação signal(). Quando a primeira thread sinaliza o semáforo, o valor é incrementado para 1 e a segunda thread é desbloqueada, podendo acessar o recurso em seguida.

7.3.1.2. Mutexes

Um mutex é uma primitiva de sincronização usada em programação concorrente para controlar o acesso a um recurso compartilhado, garantindo que apenas uma thread por vez possa acessá-lo. A palavra "mutex" é uma abreviação de "mutual exclusion", que significa exclusão mútua.

Um mutex possui dois estados possíveis: bloqueado e desbloqueado. Quando uma thread deseja acessar um recurso protegido por um mutex, ela primeiro tenta bloqueá-lo. Se o mutex estiver desbloqueado, a thread o bloqueia e pode acessar o recurso. Se o mutex já estiver bloqueado por outra thread, a thread atual é colocada em espera até que o mutex seja desbloqueado.

A operação de desbloqueio do mutex é chamada de "unlock", que é realizada quando a thread que bloqueou o mutex termina de usar o recurso compartilhado e libera o mutex para que outra thread possa acessá-lo.

A implementação de um mutex pode variar dependendo do sistema operacional e da linguagem de programação usada. Alguns sistemas operacionais fornecem mutexes como primitivas de baixo nível, enquanto em outros é possível usar bibliotecas ou APIs específicas para criar e manipular mutexes.

No entanto, a utilização incorreta de mutexes pode levar a problemas de sincronização, como deadlocks (quando duas ou mais threads ficam bloqueadas mutuamente, impedindo que a execução prossiga) e race conditions (quando duas ou mais threads tentam acessar o mesmo recurso compartilhado simultaneamente, causando resultados inesperados). Por isso, é importante que os programadores tenham um bom entendimento sobre o uso de mutexes e as práticas recomendadas para sua utilização.

7.3.1.3. Monitores

Monitores são estruturas de sincronização que garantem o acesso exclusivo a um recurso compartilhado por um conjunto de threads. Eles são implementados em linguagens de programação orientadas a objetos e consistem em uma classe que encapsula o recurso compartilhado e os métodos que podem ser usados para acessá-lo.

Os monitores possuem dois componentes principais: um bloqueio (ou lock) e uma condição (ou condition variable). O bloqueio é usado para garantir que apenas uma thread possa acessar o recurso compartilhado em um determinado momento. A condição é usada para que as threads esperem até que o recurso esteja disponível.

Quando uma thread quer acessar o recurso compartilhado, ela primeiro adquire o bloqueio do monitor. Se o recurso estiver disponível, a thread pode acessá-lo e depois liberar o bloqueio. Se o recurso estiver em uso por outra thread, a thread atual espera na condição até que o recurso esteja disponível novamente. Quando a thread que está usando o recurso o libera, ela notifica as threads que estão esperando na condição, permitindo que elas tentem acessar o recurso novamente.

Os monitores são vantajosos, pois fornecem um modelo seguro e conveniente para gerenciar o acesso a recursos compartilhados em um ambiente de programação concorrente. No entanto, eles têm algumas limitações, como a falta de suporte para comunicação explícita entre threads e a dificuldade de implementar monitores em algumas linguagens de programação. Além disso, quando uma thread está bloqueada em um monitor, ela não pode fazer nada até que o monitor seja liberado por outra thread. Isso pode levar a atrasos e possíveis problemas de deadlock se as threads esperarem indefinidamente.

7.3.1.4. Barreiras de sincronização

Barreiras de sincronização, também conhecidas como barreiras de memória, são mecanismos utilizados na programação concorrente para garantir que as operações realizadas por uma thread em um determinado ponto do programa estejam visíveis para outras threads antes que elas possam prosseguir com suas próprias operações.

Essencialmente, uma barreira de sincronização cria um ponto de sincronização no programa, no qual todas as threads que atingiram esse ponto precisam esperar até que todas as outras threads também tenham chegado nesse ponto antes que possam prosseguir. Isso é importante para garantir que todas as operações realizadas antes da barreira tenham sido concluídas antes que outras operações dependentes possam começar.

As barreiras de sincronização são especialmente úteis em cenários em que várias threads estão trabalhando em conjunto para realizar uma tarefa complexa, como processamento de imagens ou renderização gráfica. Essas tarefas geralmente envolvem várias etapas que precisam ser executadas em uma ordem específica para garantir que os resultados sejam corretos. As barreiras de sincronização garantem que cada thread execute a etapa apropriada antes que as outras possam prosseguir, o que pode ajudar a evitar erros e falhas.

Uma desvantagem das barreiras de sincronização é que elas podem levar a um atraso no desempenho, especialmente em cenários em que as threads precisam esperar por outras threads para concluir suas operações antes de prosseguir. No entanto, em muitos casos, esse atraso é necessário para garantir a integridade dos dados e a consistência dos resultados.

7.3.2. Implementação e utilização de primitivas de sincronização em diferentes linguagens de programação.

A implementação e utilização de primitivas de sincronização podem variar dependendo da linguagem de programação utilizada. Aqui estão alguns exemplos:

C/C++: A linguagem C/C++ possui bibliotecas padrão para implementar mutexes, semáforos e condições. Além disso, a biblioteca pthreads permite a criação de threads e o uso de primitivas de sincronização.

Java: A linguagem Java possui um modelo de concorrência baseado em threads e suporta sincronização através de monitor. O uso de monitor é feito através da palavra-chave "synchronized" em métodos e blocos de código.

Python: A linguagem Python possui um módulo padrão para implementar mutexes, semáforos e condições. Além disso, a biblioteca threading permite a criação de threads e o uso de primitivas de sincronização.

JavaScript: A linguagem JavaScript é executada em um ambiente de execução assíncrono e possui primitivas de sincronização como promises e async/await para lidar com operações assíncronas.

Essas são apenas algumas das linguagens de programação e suas respectivas primitivas de sincronização. Cada linguagem pode ter implementações diferentes ou adicionais para lidar com concorrência e sincronização.

7.4. Problemas clássicos de programação concorrente

Os problemas clássicos de coordenação retratam situações de coordenação entre tarefas ocorrem com muita frequência na programação de sistemas complexos, permitindo assim compreender como podem ser implementadas suas soluções. Alguns problemas clássicos são:

7.4.1. Produtores/consumidores

Descrição do problema

Este problema também é conhecido como o problema do buffer limitado, e consiste em coordenar o acesso de tarefas (processos ou threads) a um buffer compartilhado com capacidade de armazenamento limitada a N itens (que podem ser inteiros, registros, mensagens, etc.). São considerados dois tipos de processos com comportamentos cíclicos e simétricos:

Produtor: produz e deposita um item no buffer, caso o mesmo tenha uma vaga livre. Caso contrário, deve esperar até que surja uma vaga. Ao depositar um item, o produtor "consome" uma vaga livre.

Consumidor: retira um item do buffer e o consome; caso o buffer esteja vazio, aguarda que novos itens sejam depositados pelos produtores. Ao consumir um item, o consumidor "produz" uma vaga livre no buffer.

Deve-se observar que o acesso ao buffer é bloqueante, ou seja, cada processo fica bloqueado até conseguir fazer seu acesso, seja para produzir ou para consumir um item.

A solução do problema dos produtores/consumidores envolve três aspectos de coordenação distintos e complementares:

• A exclusão mútua no acesso ao buffer, para evitar condições de disputa entre produtores e/ou consumidores que poderiam corromper o conteúdo do buffer.

• A suspensão dos produtores no caso do buffer estar cheio: os produtores devem esperar até que surjam vagas livres no buffer.

• A suspensão dos consumidores no caso do buffer estar vazio: os consumidores devem esperar até que surjam novos itens a consumir no buffer.

ALGORITMO DO SEMAFORO

```
1  mutex mbuf ;          // controla o acesso ao buffer
2  semaphore item ;      // controla os itens no buffer (inicia em 0)
3  semaphore vaga ;      // controla as vagas no buffer (inicia em N)
4
5  task produtor ()
6  {
7     while (1)
8     {
9           ...                    // produz um item
10          down (vaga) ;          // espera uma vaga no buffer
11          lock (mbuf) ;          // espera acesso exclusivo ao buffer
12           ...                   // deposita o item no buffer
13           unlock (mbuf) ;       // libera o acesso ao buffer
14          up (item) ;            // indica a presença de um novo item no buffer
15    }
16 }
17
18 task consumidor
   ()
19 {
20    while (1)
21    {
22       down (item) ;     // espera um novo item no buffer
23       lock (mbuf) ;     // espera acesso exclusivo ao buffer
24       ...               // retira o item do buffer
25       unlock (mbuf) ;   // libera o acesso ao buffer
26       up (vaga) ;       // indica a liberação de uma vaga no buffer
```

```
27        ...          //          consome o item retirado do buffer
28    }
29 }
```

7.4.2. Jantar dos Filósofos

Definição do problema

Há cinco filósofos em torno de uma mesa um garfo é colocado entre cada filósofo;

Cada filósofo deve, alternadamente, refletir e comer, para que um filósofo coma, ele deve possuir dois garfos e os dois garfos devem ser aqueles logo a sua esquerda e a sua direita;

Para pegar um garfo somente pode ser pego por um filósofo e somente pode ser pego não estiver em uso por nenhum outro filósofo após comer, o filósofo deve liberar o garfo que utilizou, um filósofo pode segurar o garfo da sua direita ou o da sua esquerda assim que estiverem disponíveis mas, só pode começar a comer quando ambos estiverem sob sua posse.

A listagem a seguir é o pseudocódigo de uma implementação do comportamento básico dos filósofos, na qual cada filósofo é uma tarefa e cada palito (substituindo o garfo por hashis para os chineses) é um semáforo:

ALGORITMO DO SEMAFORO

```
1  #define NUMFILO 5
2   semaphore hashi [NUMFILO] ; // um semáforo para cada palito (iniciam em 1)
3
4   task filosofo (int i)              // filósofo i (entre 0 e 4)
5  {
6      int dir = i ;
7      int esq = (i+1) % NUMFILO ;
8
9      while (1)
```

```
10    {
11        meditar () ;
12        down (hashi [dir]) ;          // pega palito direito
13        down (hashi [esq]) ;          // pega palito esquerdo
14        comer () ;
15        up (hashi [dir]) ;            // devolve palito direito
16        up (hashi [esq]) ;            // devolve palito esquerdo
17    }
18 }
```

Como podemos ver, soluções simples para esse problema podem provocar impasses, ou seja, situações as quais todos os filósofos ficam bloqueados. Outras soluções podem provocar inanição (starvation), ou seja, alguns dos filósofos nunca conseguem comer.

Proposta de solução

Usando semáforos!

Após a estágio de pensamento tem-se ou a captura dos recursos necessário para comer ou o aguarde de uma notificação de que está apto a comer.

Diversos outros problemas clássicos são frequentemente descritos na literatura, como o problema dos leitores/escritores, do barbeiro dorminhoco, dos fumantes entre outros [Raynal, 1986; Ben-Ari, 1990].

7.5. Construções concorrentes de alto nível

Construções concorrentes de alto nível são abstrações de programação que ajudam a simplificar a criação e gerenciamento de processos e threads em um programa concorrente. Algumas das construções concorrentes de alto nível mais comuns incluem:

1. Threads: Uma thread é uma unidade básica de processamento que pode ser executada em paralelo com outras threads dentro de um mesmo processo. As threads compartilham o mesmo espaço de endereço e recursos do processo pai, o que as torna mais leves do que os processos.
2. Processos: Um processo é uma instância de um programa que pode ser executada de forma independente de outros processos. Os processos têm seu próprio espaço de endereço e recursos, o que os torna mais pesados que as threads, mas também mais isolados e seguros.
3. Canais: Um canal é uma abstração de comunicação entre processos e threads que permite que os dados sejam trocados de forma assíncrona e segura. Os canais podem ser usados para enviar e receber mensagens ou dados entre as tarefas concorrentes.
4. Semáforos: Um semáforo é uma estrutura de dados que pode ser usada para controlar o acesso concorrente a recursos compartilhados. Os semáforos são usados para garantir que apenas uma thread ou processo possa acessar um recurso compartilhado por vez.
5. Mutexes: Um mutex é uma abstração de programação que pode ser usada para sincronizar o acesso concorrente a recursos compartilhados. Os mutexes permitem que apenas uma thread ou processo possa acessar um recurso compartilhado por vez.

Essas são apenas algumas das construções concorrentes de alto nível disponíveis na programação concorrente. A escolha das construções adequadas depende do problema a ser resolvido e das características da linguagem de programação e do sistema operacional em uso. Essas construções são comumente usadas em linguagens de programação como Java, Python, C++ e outras que suportam programação concorrente.

8. Conclusão

Para o presente trabalho, o grupo concluiu que a programação concorrente é uma técnica poderosa que permite que múltiplos processos ou threads executem de forma independente, aumentando a eficiência e o desempenho do sistema. No entanto, essa técnica também pode ser desafiadora, pois introduz problemas de sincronização e comunicação entre os processos, que precisam ser resolvidos com o uso de primitivas de sincronização e construções concorrentes de alto nível. Portanto, é essencial para os programadores compreenderem bem os conceitos de programação concorrente e as primitivas de sincronização disponíveis nas linguagens de programação, para evitar problemas como condições de corrida, deadlock e starvation. Com a crescente demanda por sistemas escaláveis e eficientes, a programação concorrente se tornou uma habilidade essencial para os desenvolvedores, e continuar a aprimorar essa técnica é fundamental para se manter competitivo no mercado de tecnologia.

9. Referências bibliográficas

M. Herlihy, N. Shavit.The Art of Multiprocessor Programming (TAMP), Morgan Kauffman Publishers, 2008.

DAVIS, William S. Sistemas operacionais: uma visão sistemática. 9.ed. Rio de Janeiro: Campus, 1991.

Woodhull, Albert S., Tanenbaum, Andrew S. Sistemas Operacionais: Projecto e implementação. 3.ed. Porto Alegre: Bookman, 2008.

Maziero, Carlos. Sistemas Operacionais: Conceitos e Mecanismos. 2.ed.Curitiba: UFPR, 2019.

Barros, D., Peres, R. Threads, semáforos e deadlocks – o jantar dos filósofos. Revista PROGRAMAR. 2013, Fevereiro 26. – artigo.