UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ
CIÊNCIA DA COMPUTAÇÃO DISCIPLINA DE SISTEMAS OPERACIONAIS

Relatório - Trabalho 1

Sistemas Operacionais

Leticia Zanellatto de Oliveira

Moises Sobha Fayad

Foz do Iguaçu

2024

Sumário

1 INTRODUÇÃO

O problema do produtor-consumidor é um dos problemas clássicos de sincronização na computação e representa um desafio fundamental na construção de sistemas concorrentes e paralelos. Esse problema surge em contextos onde há uma ou mais entidades (produtores) que geram dados ou tarefas e outras entidades (consumidores) que processam esses dados. Ambos, produtor e consumidor, precisam acessar um recurso compartilhado, geralmente um buffer, para armazenar e processar esses itens de forma ordenada e eficiente (1).

Em sistemas reais, o problema do produtor-consumidor é amplamente aplicado em filas de tarefas, sistemas de processamento de dados em lote, servidores de fila de mensagens, e até mesmo no funcionamento interno de sistemas operacionais. Por exemplo, em sistemas de produção e consumo de dados, como os pipelines de processamento, os dados produzidos precisam ser consumidos em tempo hábil para garantir o fluxo contínuo do sistema, evitando sobrecargas e subutilização de recursos (2).

A complexidade desse problema está na sincronização entre as operações de produção e consumo. Sem mecanismos de controle adequados, o produtor pode tentar inserir novos itens em um buffer cheio, ou o consumidor pode tentar consumir itens de um buffer vazio, o que leva a erros de execução ou ineficiência no sistema. Para resolver esse problema, é necessário implementar mecanismos de controle de acesso ao buffer, como semáforos, locks ou filas de mensagens, que permitam a coordenação entre produtores e consumidores.

O objetivo deste trabalho é estudar o problema do produtor-consumidor e implementar uma solução que permita o gerenciamento eficiente dos acessos ao buffer compartilhado, garantindo a sincronização e o funcionamento correto do sistema.

2 SEMAFRO

Este trabalho explora uma solução para o problema usando semáforos e sincronização de threads em Java. A abordagem implementa mecanismos de controle de acesso ao buffer compartilhado entre produtores e consumidores, assegurando que ambos operem de maneira ordenada e evitando conflitos de acesso. A solução proposta utiliza os recursos da classe `Semaphore` de Java, do pacote `java.util.concurrent`. A escolha da linguagem e dos recursos foi motivada pela eficiência e simplicidade da API de concorrência do Java, que facilita a sincronização de threads e o controle de recursos compartilhados (3).

A implementação apresentada resolve o problema do produtor-consumidor utilizando semáforos para controlar o acesso a um buffer compartilhado. Esta solução, em Java, faz uso de semáforos e threads para permitir que múltiplos produtores e consumidores operem simultaneamente sem conflitos, garantindo a integridade dos dados e a sincronização dos acessos ao buffer.

2.1 Buffer e Semáforos

A classe `Semafro` contém:

- **Buffer**: O buffer é representado por um array de inteiros de tamanho `bufferSize`, onde os produtores inserem itens e os consumidores os removem.
- **Semáforos**: São utilizados três semáforos para o controle de acesso:

- `empty`: Conta os espaços vazios no buffer e é inicializado com o tamanho total do buffer. Controla a disponibilidade de espaços para inserção.
- `full`: Conta os itens disponíveis para consumo, iniciando com zero. Controla a disponibilidade de itens no buffer.
- `mutex`: Um semáforo binário que garante acesso exclusivo ao buffer durante operações de inserção e remoção, prevenindo condições de corrida.

2.2 Métodos de Produção e Consumo

- **Método de Inserção** (`insert`): Usado pelos produtores para adicionar itens ao buffer. O método aguarda a disponibilidade de espaço (`empty.acquire()`) e bloqueia o acesso exclusivo ao buffer (`mutex.acquire()`) antes de inserir o item e atualizar o índice de inserção (`in`). Após a inserção, libera o `mutex` e incrementa o semáforo `full` para indicar um item disponível.
- **Método de Remoção** (`remove`): Usado pelos consumidores para remover itens do buffer. O método aguarda que haja um item disponível (`full.acquire()`), bloqueia o acesso exclusivo (`mutex.acquire()`), remove o item e atualiza o índice de remoção (`out`). Após a remoção, libera o `mutex` e incrementa `empty` para indicar que há espaço livre.

2.3 Classes Produtor e Consumidor

- **Produtor**: A classe `Producer` implementa a produção de itens. Ela executa um loop infinito, onde produz um item (número aleatório entre 0 e 4), insere-o no buffer e espera um tempo (`productionTime`) antes de produzir o próximo item.
- **Consumidor**: A classe `Consumer` implementa o consumo de itens. Ela executa um loop infinito, removendo um item do buffer, simulando o consumo do item e aguardando um tempo (`consumptionTime`) antes de consumir o próximo item.

2.4 Funcionamento da Sincronização

- **Sincronização**: Os semáforos `empty` e `full` garantem que os produtores só inserem itens quando há espaço no buffer e que os consumidores só removem quando há itens disponíveis. O semáforo `mutex` controla o acesso exclusivo ao buffer durante as operações de inserção e remoção, prevenindo condições de corrida.
- **Concorrência**: Múltiplas threads de produtores e consumidores podem operar em paralelo, respeitando as permissões dos semáforos e evitando interferência direta, o que permite a sincronização eficiente entre várias threads.

2.5 Revisão da Literatura

A teoria dos semáforos, introduzida por Edsger Dijkstra, é uma base consolidada na área de programação concorrente e sistemas operacionais. No contexto acadêmico, Dijkstra (1968) apresenta o conceito de semáforos como uma ferramenta essencial para resolver problemas de sincronização e comunicação entre processos cooperantes. Semáforos binários e contadores são amplamente utilizados em problemas de controle de acesso, sendo aplicados em soluções de sistemas reais como filas de processamento e gestão de recursos(4).

Tanenbaum e Bos (2014), em *Modern Operating Systems*, ampliam essa discussão, abordando como semáforos e outros mecanismos de sincronização são aplicados para garantir o funcionamento correto de sistemas multiprogramados. A escolha pelo uso de semáforos é justificada por sua eficácia em resolver problemas de acesso concorrente e pela flexibilidade em ajustar o controle de permissões, essencial para a solução de problemas como o produtor-consumidor(1).

```
#define N 100                                   /* numero de lugares no buffer */
typedef int semaphore;                          /* semaforos sao um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso a regiao critica */
semaphore empty = N;                            /* conta os lugares vazios no buffer */
semaphore full = 0;                             /* conta os lugares preenchidos no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE e a constante 1 */
          item = produce_item();                /* gera algo para por no buffer */
          down(&empty);                         /* decresce o contador empty */
          down(&mutex);                         /* entra na regiao critica */
          insert_item(item);                    /* poe novo item no buffer */
          up(&mutex);                           /* sai da regiao critica */
          up(&full);                            /* incrementa o contador de lugares preenchidos */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* laco infinito */
          down(&full);                          /* decresce o contador full */
          down(&mutex);                         /* entra na regiao critica */
          item = remove_item();                 /* pega item do buffer */
          up(&mutex);                           /* sai da regiao critica */
          up(&empty);                           /* incrementa o contador de lugares vazios */
          consume_item(item);                   /* faz algo com o item */
     }
}
```

Figura 1: Implementação do Semafro (1).

2.6 Justificativa da Implementação

A implementação garante que o buffer compartilhado seja acessado de maneira segura e sincronizada, mesmo com múltiplos produtores e consumidores atuando simultaneamente. O uso de semáforos é ideal para controlar a quantidade de acessos e garantir que o sistema funcione sem sobrecarga de recursos ou perda de dados. Essa abordagem permite que o sistema funcione corretamente em um ambiente concorrente, evitando problemas como deadlocks e condições de corrida.

3 SOCKET

O uso de passagem de mensagens é uma abordagem comum e eficaz em sistemas distribuídos para resolver problemas de comunicação e sincronização entre processos concorrentes, como o problema do produtor-consumidor. Diferente de abordagens que utilizam buffers compartilhados ou mecanismos de controle de acesso direto ao recurso, a passagem de mensagens permite que cada processo, seja ele um produtor ou consumidor, comunique-se de maneira isolada, utilizando apenas canais de troca de informações. Esse método reduz a necessidade de controle explícito sobre o acesso ao recurso, garantindo que o produtor e o consumidor se comuniquem diretamente sem interferência externa (5). Uma das abordagens para resolver este problema é o uso de **sockets**, pois através dele é permitido uma comunicação eficiente e síncrona entre threads ou processos, possibilitando o envio e recebimento de dados sem a necessidade de um buffer compartilhado.

Os sockets são uma interface de comunicação de baixo nível que possibilita a troca de mensagens entre dispositivos, seja na mesma máquina ou em máquinas distintas conectadas por uma rede. Essa técnica é amplamente utilizada em sistemas distribuídos, onde o compartilhamento de recursos e a comunicação entre processos são fundamentais (6).

3.1 Solução utilizando Sockets para Produtor-Consumidor

Nesta solução, implementa-se um sistema onde os produtores e consumidores são threads separadas que comunicam-se via sockets. O produtor envia mensagens (itens produzidos) para o consumidor através de um socket, enquanto o consumidor lê essas mensagens e as processa. Esse modelo é uma alternativa ao uso de buffers compartilhados, garantindo a sincronização e permitindo que o sistema seja escalável e facilmente distribuído.

3.2 Estrutura da Solução

- **Socket do Produtor**: Cada produtor possui um socket cliente que se conecta a um socket servidor mantido pelo consumidor. Quando o produtor gera um item, ele envia uma mensagem através do socket para o consumidor, sinalizando a disponibilidade do item.
- **Socket do Consumidor**: O consumidor cria um socket servidor que aguarda conexões dos produtores. Ao receber uma conexão, o consumidor lê a mensagem contendo o item produzido e o consome. Após o processamento, o consumidor envia uma mensagem de confirmação de recebimento, permitindo que o produtor envie um novo item.
- **Controle de Fluxo**: Para evitar que o consumidor tente ler de um socket sem mensagens ou que o produtor sobrecarregue o consumidor, são utilizadas mensagens de controle. O consumidor envia uma mensagem vazia após cada item consumido, sinalizando ao produtor que ele está pronto para receber o próximo item.

3.3 Lógica da Comunicação

- **1** O consumidor abre um socket servidor e fica em modo de escuta, aguardando conexões de um ou mais produtores.
- **2** Quando o produtor conecta ao servidor, ele envia uma mensagem com o item produzido.
- **3** O consumidor lê a mensagem, processa o item e então envia uma mensagem de confirmação, sinalizando que está pronto para receber o próximo item.
- **4** O produtor, ao receber a confirmação, gera o próximo item e repete o processo.

Esse modelo pode ser facilmente escalado para múltiplos consumidores, utilizando-se múltiplos sockets ou gerenciando as conexões com uma fila de clientes. O uso de sockets para comunicação também permite que o sistema seja distribuído, operando em diferentes dispositivos conectados em rede.

3.4 Vantagens do Uso de Sockets

O uso de sockets para resolver o problema do produtor-consumidor oferece vantagens significativas:

- **Distribuição**: O sistema pode operar em múltiplos dispositivos ou máquinas, permitindo escalabilidade e flexibilidade.
- **Independência de Buffer**: Não há necessidade de buffer compartilhado, o que simplifica a implementação e reduz a possibilidade de condições de corrida.
- **Controle Direto de Comunicação**: O uso de mensagens para controle de fluxo permite um controle preciso da comunicação entre produtor e consumidor, evitando sobrecarga.

3.5 Revisão da Literatura

Tanenbaum descreve a passagem de mensagens como uma técnica fundamental para a comunicação entre processos em sistemas distribuídos, onde os processos geralmente operam em máquinas distintas e precisam coordenar suas atividades. Nesse modelo, o processo consumidor age como servidor, aguardando conexões e mensagens dos processos produtores, que se conectam como clientes. O consumidor lê cada mensagem enviada pelos produtores, processa o item contido na mensagem e responde ao produtor confirmando o recebimento e processamento. Essa troca de mensagens bidirecional garante que o produtor não sobrecarregue o consumidor com itens antes que ele esteja pronto para processar o próximo. Além disso, permite que o sistema seja distribuído em diferentes máquinas, aumentando a flexibilidade e a eficiência da comunicação (1).

```
#define N 100                                   /* numero de lugares no buffer */

void producer(void)
{
     int item;
     message m;                                 /* buffer de mensagens */

     while (TRUE) {
          item = produce_item( );               /* gera alguma coisa para colocar no buffer */
          receive(consumer, &m);                /* espera que uma mensagem vazia chegue */
          build_message(&m, item);              /* monta uma mensagem para enviar */
          send(consumer, &m);                   /* envia item para consumidor */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* envia N mensagens vazias */
     while (TRUE) {
          receive(producer, &m);                /* pega mensagem contendo item */
          item = extract_item(&m);              /* extrai o item da mensagem */
          send(producer, &m);                   /* envia a mensagem vazia como resposta */
          consume_item(item);                   /* faz alguma coisa com o item */
     }
}
```

Figura 2: O problema produtor-consumidor com N mensagens (1).

3.6 Justificativa

A utilização de sockets para implementar o problema do produtor-consumidor é uma solução eficiente para ambientes distribuídos, onde a comunicação direta entre processos é essencial. Essa abordagem elimina a necessidade de buffer compartilhado, permitindo que produtores e consumidores operem em máquinas separadas e garantam a sincronização de forma segura. Além disso, a escalabilidade oferecida pelos sockets é ideal para sistemas que demandam flexibilidade e crescimento.

6 REFERÊNCIAS BIBLIOGRÁFICAS

1 TANENBAUM, A.; BOS, H. *Modern Operating Systems*. 4th. ed. [S.l.]: Pearson, 2014.

2 SILBERSCHATZ, A.; GALVIN, P.; GAGNE, G. *Operating System Concepts*. 9th. ed. [S.l.]: Wiley, 2013.

3 Oracle Documentation. *Class Semaphore*. 2023. `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html`. Disponível em: <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html>.

4 DIJKSTRA, E. Cooperating sequential processes. In: GENUYS, F. (Ed.). *Programming Languages: NATO Advanced Study Institute, 1967*. [S.l.]: Academic Press, 1968. p. 43–112.

5 TANENBAUM, A. S.; STEEN, M. V. *Distributed Systems: Principles and Paradigms*. 2nd. ed. [S.l.]: Pearson, 2007.

6 Oracle Documentation. *Class Socket*. 2023. `https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html`. Disponível em: <https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html>.