UNIVERSIDADE DO ESTADO DO RIO DE JANEIRO

MESTRADO EM CIÊNCIAS COMPUTACIONAIS

SISTEMAS OPERACIONAIS

SOLUÇÃO DO PROBLEMA DE PRODUTORES E CONSUMIDORES UTILIZANDO REDES DE PETRI

Matheus Zeitune

Rio de Janeiro

2025

RESUMO

Este trabalho apresenta uma solução completa para o problema clássico de produtores e consumidores utilizando Redes de Petri. A modelagem proposta implementa um sistema com 2 produtores e 2 consumidores operando sobre um buffer compartilhado, utilizando dois semáforos (item e vaga) para garantir a sincronização adequada. A solução permite maior concorrência ao possibilitar o acesso simultâneo de um produtor e um consumidor ao buffer. Adicionalmente, analisa-se criticamente o impacto da inversão da ordem de aquisição dos semáforos, demonstrando-se como essa alteração pode levar a situações de deadlock. Implementações em linguagem C de ambos os protocolos são fornecidas, permitindo verificação prática dos conceitos teóricos. A verificação do modelo por meio de simulação em ferramenta especializada comprova a corretude da solução proposta e valida as análises teóricas apresentadas.

Palavras-chave: Redes de Petri. Produtores e Consumidores. Sincronização. Semáforos. Concorrência. Deadlock. POSIX Threads.

SUMÁRIO

1. INTRODUÇÃO
2. FUNDAMENTAÇÃO TEÓRICA
2.1 O Problema de Produtores e Consumidores
2.2 Redes de Petri
2.3 Semáforos e Sincronização
3. SOLUÇÃO PROPOSTA COM REDES DE PETRI
3.1 Modelagem com Redes de Petri
3.2 Configuração dos Semáforos
3.3 Aumento da Concorrência
4. IMPLEMENTAÇÃO EM C - PROTOCOLO CORRETO
4.1 Código Completo
4.2 Análise do Código
5. ANÁLISE DA INVERSÃO DE SEMÁFOROS
5.1 Protocolo Original
5.2 Protocolo com Inversão
5.3 Caracterização do Deadlock
6. IMPLEMENTAÇÃO EM C - PROTOCOLO INVERTIDO
6.1 Código com Inversão
6.2 Demonstração do Deadlock
7. RESULTADOS E DISCUSSÃO
8. CONCLUSÃO
REFERÊNCIAS
APÊNDICE A - ARQUIVO XML DA REDE DE PETRI (CORRETO)
APÊNDICE B - ARQUIVO XML DA REDE DE PETRI (INVERTIDO)
APÊNDICE C - INSTRUÇÕES DE COMPILAÇÃO E EXECUÇÃO

1. INTRODUÇÃO

O problema de produtores e consumidores é um dos problemas clássicos de sincronização em sistemas operacionais, originalmente proposto por Edsger Dijkstra na década de 1960. Este problema modela uma situação comum em sistemas concorrentes, onde múltiplos processos produtores geram dados e os depositam em um buffer compartilhado, enquanto processos consumidores retiram esses dados para processamento.

A complexidade deste problema reside na necessidade de coordenação entre os processos para evitar condições de corrida, garantir exclusão mútua quando necessário e prevenir situações de deadlock. Tradicionalmente, a solução deste problema utiliza semáforos como mecanismos de sincronização, implementando protocolos específicos para controle de acesso ao buffer compartilhado.

As Redes de Petri constituem um formalismo matemático e gráfico poderoso para modelagem, análise e simulação de sistemas concorrentes, distribuídos e assíncronos. Desenvolvidas por Carl Adam Petri em 1962, estas redes permitem a representação visual da dinâmica de sistemas, facilitando a compreensão de interações complexas entre componentes concorrentes.

Este trabalho apresenta uma solução completa para o problema de produtores e consumidores utilizando três abordagens complementares: modelagem formal com Redes de Petri, análise teórica dos protocolos de sincronização e implementação prática em linguagem C utilizando POSIX threads. A solução proposta utiliza dois semáforos (item e vaga) e permite maior grau de concorrência ao possibilitar que um produtor e um consumidor acessem simultaneamente o buffer. Adicionalmente, investiga-se o comportamento do sistema quando a ordem de aquisição dos semáforos é invertida, tanto através de modelagem formal quanto de implementação prática, demonstrando-se a ocorrência de deadlock em ambos os casos.

2. FUNDAMENTAÇÃO TEÓRICA

2.1 O Problema de Produtores e Consumidores

O problema de produtores e consumidores modela uma situação onde processos produtores geram itens de dados e os depositam em um buffer de capacidade finita, enquanto processos consumidores retiram esses itens para processamento. Os desafios principais incluem:

- Sincronização: produtores devem aguardar quando o buffer está cheio, e consumidores devem aguardar quando o buffer está vazio.
- Exclusão mútua: o acesso ao buffer deve ser protegido para evitar condições de corrida.
- Ausência de deadlock: o protocolo de sincronização não deve permitir situações onde todos os processos ficam bloqueados indefinidamente.

2.2 Redes de Petri

Uma Rede de Petri é formalmente definida como uma quádrupla $RP = (P, T, F, M_0)$, onde P é um conjunto finito de lugares (places), T é um conjunto finito de transições (transitions), $F \subseteq (P \times T) \cup (T \times P)$ é o conjunto de arcos direcionados, e $M_0 : P \rightarrow \mathbb{N}$ é a marcação inicial que atribui tokens aos lugares.

A dinâmica das Redes de Petri é governada por regras de disparo de transições. Uma transição t está habilitada se todos os seus lugares de entrada contêm pelo menos um token. Quando uma transição habilitada dispara, ela consome um token de cada lugar de entrada e produz um token em cada lugar de saída, representando a evolução do estado do sistema.

Para modelagem de sistemas com semáforos, as Redes de Petri oferecem uma representação natural onde lugares representam estados ou recursos, transições representam operações ou eventos, e tokens representam a disponibilidade de recursos ou a presença em determinado estado.

2.3 Semáforos e Sincronização

Semáforos são mecanismos de sincronização propostos por Dijkstra que consistem em uma variável inteira não-negativa e duas operações atômicas: wait() e signal(). A operação wait(S) decrementa o semáforo S e bloqueia o processo se $S < 0$, enquanto signal(S) incrementa S e pode desbloquear um processo em espera.

No problema de produtores e consumidores, semáforos são utilizados para coordenação:

- Semáforo 'item': controla o número de itens disponíveis no buffer para consumo, inicializado em 0.
- Semáforo 'vaga': controla o número de posições vazias no buffer, inicializado com a capacidade do buffer.

A ordem correta de aquisição destes semáforos é crucial para evitar deadlock. O protocolo padrão estabelece que produtores devem primeiro aguardar por uma vaga

disponível, depois adquirir exclusão mútua se necessário, depositar o item, liberar a exclusão mútua e finalmente sinalizar a disponibilidade de um novo item.

3. SOLUÇÃO PROPOSTA COM REDES DE PETRI

3.1 Modelagem com Redes de Petri

A modelagem proposta representa o sistema completo com 2 produtores e 2 consumidores operando sobre um buffer compartilhado de capacidade 5. Cada produtor e consumidor é modelado como um ciclo independente de operações, interagindo com os recursos compartilhados por meio de transições que representam as operações de sincronização.

Os principais componentes da Rede de Petri incluem:

- Lugares representando estados dos produtores: P1_Produzindo, P1_Aguardando_Vaga, P1_Depositando (e equivalentes para P2).
- Lugares representando estados dos consumidores: C1_Aguardando_Item, C1_Retirando, C1_Consumindo (e equivalentes para C2).
- Lugares representando os semáforos: Vagas_Disponíveis (5 tokens iniciais) e Itens_Disponíveis (0 tokens iniciais).
- Transições representando operações wait() e signal() sobre os semáforos, bem como operações de produção e consumo.

3.2 Configuração dos Semáforos

A solução utiliza dois semáforos principais para coordenação:

- Semáforo 'vaga': inicializado com 5 tokens, representa vagas disponíveis no buffer.
- Semáforo 'item': inicializado com 0 tokens, representa itens presentes no buffer.

Protocolo do Produtor:

wait(vaga);

<depositar item>

signal(item);

Protocolo do Consumidor:

wait(item);

<retirar item>

signal(vaga);

3.3 Aumento da Concorrência

A solução permite maior concorrência ao eliminar a necessidade de um semáforo mutex global. Quando o buffer é implementado como uma estrutura que suporta operações concorrentes de inserção e remoção (como uma fila circular com ponteiros independentes), um produtor e um consumidor podem operar simultaneamente, maximizando o throughput do sistema. Esta característica é naturalmente representada nas Redes de Petri pela possibilidade de múltiplas transições dispararem independentemente.

4. IMPLEMENTAÇÃO EM C - PROTOCOLO CORRETO

4.1 Código Completo

A implementação em linguagem C utiliza POSIX threads (pthread) e semáforos POSIX para demonstrar o funcionamento correto do protocolo:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5
#define NUM_PRODUTORES 2
#define NUM_CONSUMIDORES 2
#define NUM_ITENS 20

int buffer[BUFFER_SIZE];
int in = 0, out = 0;

sem_t vaga, item;
pthread_mutex_t mutex;


void* produtor(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < NUM_ITENS / NUM_PRODUTORES; i++) {
        int item = rand() % 100;
        printf("Produtor %d: produziu %d\n", id, item);

        sem_wait(&vaga);  // Aguarda vaga
        pthread_mutex_lock(&mutex);
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&item);  // Sinaliza item

        usleep(100000);
    }
    return NULL;
}

void* consumidor(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < NUM_ITENS / NUM_CONSUMIDORES; i++) {
        sem_wait(&item);  // Aguarda item
        pthread_mutex_lock(&mutex);
        int item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&vaga);  // Sinaliza vaga
```

```
            printf("Consumidor %d: consumiu %d\n", id, item);
            usleep(150000);
        }
        return NULL;
    }
```

4.2 Análise do Código

A implementação segue rigorosamente o protocolo correto. O produtor executa wait(vaga) antes de depositar, garantindo que existe espaço disponível. Após o depósito, executa signal(item) para notificar os consumidores. O consumidor, complementarmente, executa wait(item) para aguardar disponibilidade, retira o item e executa signal(vaga) para liberar espaço. Esta ordem elimina a possibilidade de deadlock e garante progresso contínuo do sistema.

5. ANÁLISE DA INVERSÃO DE SEMÁFOROS

5.1 Protocolo Original

No protocolo correto, a ordem wait(vaga) → deposita → signal(item) para produtores e wait(item) → retira → signal(vaga) para consumidores garante que não ocorrerá deadlock. A correção pode ser verificada observando-se que a soma dos tokens em Vagas_Disponíveis e Itens_Disponíveis permanece constante e igual à capacidade do buffer.

5.2 Protocolo com Inversão

Quando a ordem é invertida, o produtor passa a executar:

wait(item); // ERRADO! Deveria ser wait(vaga)
<depositar>
signal(vaga); // ERRADO! Deveria ser signal(item)

Esta inversão cria uma contradição lógica: o produtor aguarda por um item antes de produzir um item.

5.3 Caracterização do Deadlock

Com buffer inicialmente vazio (0 itens, 5 vagas):

- Produtores tentam wait(item), mas item = 0, então todos bloqueiam.
- Consumidores também tentam wait(item) e bloqueiam.
- Nenhum processo pode progredir → DEADLOCK!

Esta situação satisfaz as quatro condições de Coffman: exclusão mútua, posse e espera, não-preempção e espera circular.

6. IMPLEMENTAÇÃO EM C - PROTOCOLO INVERTIDO

6.1 Código com Inversão

Modificando apenas a função produtor para inverter a ordem:

```
void* produtor(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < NUM_ITENS / NUM_PRODUTORES; i++) {
        int item = rand() % 100;
        printf("Produtor %d: produziu %d\n", id, item);

        // INVERTIDO:
        sem_wait(&item);   // ERRADO! wait(item)
        pthread_mutex_lock(&mutex);
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&vaga);   // ERRADO! signal(vaga)

        usleep(100000);
    }
    return NULL;
}
```

6.2 Demonstração do Deadlock

Ao executar este código, observa-se:

```
=== PROTOCOLO INVERTIDO (DEADLOCK!) ===
Produtor 1: produziu 83
Produtor 2: produziu 47
[PROGRAMA TRAVA - todos bloqueados em wait(item)]
```

O programa nunca termina. Todos os processos (produtores e consumidores) ficam bloqueados aguardando pelo semáforo item que está em 0 e nunca será incrementado, pois nenhum processo consegue progredir. É necessário usar Ctrl+C para interromper a execução.

7. RESULTADOS E DISCUSSÃO

A implementação prática em C e a modelagem com Redes de Petri confirmam mutuamente os resultados teóricos:

- **Protocolo correto:** Tanto na simulação da Rede de Petri quanto na execução do código C, o sistema progride continuamente. Produtores e consumidores alternam, o invariante estrutural é mantido, e não há estados mortos alcançáveis.
- **Protocolo invertido:** Em ambas as abordagens, o deadlock ocorre imediatamente. Na Rede de Petri, nenhuma transição fica habilitada. No código C, todos os threads ficam bloqueados em sem_wait(&item).

A comparação evidencia a importância crítica da ordem de aquisição de recursos. Uma modificação aparentemente simples transforma um sistema funcional em um sistema que falha imediatamente. As Redes de Petri demonstram-se uma ferramenta valiosa para análise formal, enquanto a implementação em C valida os resultados na prática e permite observação do comportamento em um sistema real.

Em termos de desempenho, a solução com maior concorrência oferece potencial para maior throughput. A verificação por simulação e por execução prática demonstra que o sistema pode operar com um produtor e um consumidor simultaneamente, maximizando a utilização do buffer.

8. CONCLUSÃO

Este trabalho apresentou uma solução completa para o problema clássico de produtores e consumidores utilizando três abordagens complementares: modelagem formal com Redes de Petri, análise teórica dos protocolos de sincronização e implementação prática em linguagem C com POSIX threads.

A principal contribuição é demonstrar como diferentes formalismos convergem para os mesmos resultados. As Redes de Petri permitiram verificação formal das propriedades de correção, a análise teórica fundamentou o entendimento dos mecanismos de sincronização, e a implementação em C validou os conceitos na prática.

A análise da inversão da ordem dos semáforos demonstrou de forma inequívoca as consequências desta modificação. O sistema que funcionava perfeitamente torna-se imediatamente propenso a deadlock. Esta análise comparativa, realizada tanto em Redes de Petri quanto em código executável, serve como exemplo pedagógico valioso sobre a importância da ordem de aquisição de recursos em sistemas concorrentes.

As ferramentas utilizadas - Redes de Petri para modelagem visual e análise formal, e POSIX threads para implementação prática - provaram-se complementares e essenciais para compreensão completa do problema. A combinação destas abordagens oferece ao estudante de Sistemas Operacionais uma visão abrangente desde a teoria até a prática.

Como trabalhos futuros, sugere-se a extensão do modelo para considerar políticas de prioridade entre processos, análise de desempenho quantitativo utilizando Redes de Petri Estocásticas, e implementação de variações do problema como o jantar dos filósofos e leitores-escritores.

REFERÊNCIAS

COFFMAN, E. G.; ELPHICK, M.; SHOSHANI, A. System deadlocks. ACM Computing Surveys, v. 3, n. 2, p. 67-78, 1971.

DIJKSTRA, E. W. Cooperating sequential processes. In: GENUYS, F. (Ed.). Programming Languages. London: Academic Press, 1968. p. 43-112.

MURATA, T. Petri nets: Properties, analysis and applications. Proceedings of the IEEE, v. 77, n. 4, p. 541-580, 1989.

PETERSON, J. L. Petri Net Theory and the Modeling of Systems. Englewood Cliffs: Prentice-Hall, 1981.

REISIG, W. Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies. Berlin: Springer, 2013.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. Operating System Concepts. 10. ed. New York: John Wiley & Sons, 2018.

TANENBAUM, A. S.; BOS, H. Modern Operating Systems. 4. ed. Upper Saddle River: Pearson, 2015.

APÊNDICE A

ARQUIVO XML DA REDE DE PETRI (PROTOCOLO CORRETO)

O arquivo XML a seguir deve ser salvo como produtores_consumidores.xml e pode ser importado em ferramentas de simulação de Redes de Petri como PIPE, CPN Tools ou WoPeD:

Características da rede:

- 2 produtores e 2 consumidores
- Buffer de tamanho 5 (Vagas_Disponíveis = 5 tokens iniciais)
- Ordem correta: wait(vaga) → deposita → signal(item)
- Sistema livre de deadlock

Observação: O arquivo XML completo está disponível em produtores_consumidores.xml. Para visualizá-lo, abra o arquivo em um editor de texto ou importe-o diretamente em uma ferramenta de simulação.

APÊNDICE B

ARQUIVO XML DA REDE DE PETRI (PROTOCOLO INVERTIDO)

O arquivo XML a seguir deve ser salvo como produtores_consumidores_invertido_deadlock.xml e demonstra o comportamento do sistema com ordem invertida dos semáforos:

Características da rede:

- Ordem INVERTIDA: wait(item) → deposita → signal(vaga)
- Deadlock imediato na marcação inicial
- Uso didático para demonstrar importância da ordem

Observação: O arquivo XML completo está disponível em produtores_consumidores_invertido_deadlock.xml. Ao simular esta rede, observe que nenhuma transição fica habilitada após a marcação inicial.

APÊNDICE C

INSTRUÇÕES DE COMPILAÇÃO E EXECUÇÃO

Compilação dos Programas em C

Os programas requerem GCC e a biblioteca pthread (POSIX threads). Para compilar:

```
# Compilar protocolo correto
gcc -o prodcons_correto prodcons_correto.c -lpthread

# Compilar protocolo invertido
gcc -o prodcons_invertido prodcons_invertido.c -lpthread
```

Execução

```
# Executar protocolo correto (funciona normalmente)
./prodcons_correto

# Executar protocolo invertido (DEADLOCK!)
./prodcons_invertido
# Use Ctrl+C para interromper
```

Ferramentas para Simulação de Redes de Petri

Recomenda-se o uso das seguintes ferramentas:

- **PIPE (Platform Independent Petri Net Editor)** - Ferramenta Java de código aberto - http://pipe2.sourceforge.net/
- **CPN Tools** - Para Redes de Petri Coloridas - https://cpntools.org/
- **WoPeD (Workflow Petri Net Designer)** - Ferramenta educacional - https://woped.dhbw-karlsruhe.de/

Evidências de Execução

Para documentar adequadamente a execução:

- Capture screenshots da simulação em Redes de Petri mostrando estados inicial e final
- Salve a saída dos programas C em arquivos de texto para análise
- Compare os comportamentos entre protocolo correto e invertido