UNIVERSIDADE DO ESTADO DO RIO DE JANEIRO
MESTRADO EM CIÊNCIAS COMPUTACIONAIS
SISTEMAS OPERACIONAIS

SOLUÇÃO DO PROBLEMA DO JANTAR DOS FILÓSOFOS UTILIZANDO INVERSÃO DE GARFOS DOS FILÓSOFOS PARES

Matheus Zeitune

Rio de Janeiro
2025

RESUMO

Este trabalho apresenta uma solução completa para o problema clássico do Jantar dos Filósofos utilizando Redes de Petri, com foco na Solução 4, que consiste em inverter a ordem de aquisição dos garfos para os filósofos em posições pares. A modelagem proposta implementa um sistema com 5 filósofos e 5 garfos, onde os filósofos ímpares (1, 3, 5) pegam primeiro o garfo esquerdo e depois o direito, enquanto os filósofos pares (2, 4) pegam primeiro o garfo direito e depois o esquerdo. Esta assimetria na ordem de aquisição quebra o ciclo de espera circular, eliminando a possibilidade de deadlock. O trabalho apresenta análise detalhada das propriedades de segurança (exclusão mútua) e vivacidade (ausência de deadlock), além de investigar cenários de starvation e propor soluções. Adicionalmente, discute-se a representação de semáforos FIFO em Redes de Petri tradicionais e suas extensões. Implementações em linguagem C utilizando POSIX threads são fornecidas para validação prática dos conceitos teóricos.

Palavras-chave: Redes de Petri. Jantar dos Filósofos. Sincronização. Deadlock. Starvation. Semáforos. POSIX Threads.

SUMÁRIO

1. INTRODUÇÃO

O problema do Jantar dos Filósofos é um dos problemas clássicos de sincronização em sistemas operacionais, proposto por Edsger Dijkstra em 1965. Este problema modela situações onde múltiplos processos competem por recursos compartilhados limitados, sendo fundamental para o entendimento de conceitos como deadlock, starvation e sincronização em sistemas concorrentes.

O problema clássico apresenta cinco filósofos sentados ao redor de uma mesa circular, com um garfo entre cada par de filósofos. Cada filósofo alterna entre pensar e comer, mas para comer precisa de dois garfos (o da esquerda e o da direita). A complexidade reside na necessidade de coordenação entre os filósofos para evitar deadlock (todos ficam bloqueados aguardando recursos) e starvation (algum filósofo nunca consegue comer).

Existem diversas soluções propostas para este problema, cada uma com diferentes características e trade-offs. A Solução 4, foco deste trabalho, consiste em inverter a ordem de aquisição dos garfos para os filósofos em posições pares. Enquanto filósofos ímpares pegam primeiro o garfo esquerdo e depois o direito, filósofos pares fazem o inverso. Esta assimetria quebra o ciclo de espera circular, eliminando a possibilidade de deadlock.

Este trabalho apresenta uma análise completa da Solução 4 utilizando três abordagens complementares: modelagem formal com Redes de Petri, análise teórica das propriedades de segurança e vivacidade, e implementação prática em linguagem C utilizando POSIX threads. Adicionalmente, investiga-se o problema de starvation, propondo-se uma solução baseada em controle de prioridade, e discute-se a representação de semáforos FIFO em Redes de Petri e suas extensões.

2. FUNDAMENTAÇÃO TEÓRICA

2.1 O Problema do Jantar dos Filósofos

O problema do Jantar dos Filósofos modela uma situação de competição por recursos compartilhados. Cinco filósofos sentam-se ao redor de uma mesa circular, com um prato de espaguete à frente de cada um e um garfo entre cada par de filósofos. Um filósofo precisa de ambos os garfos (esquerdo e direito) para comer.

Cada filósofo executa o seguinte ciclo indefinidamente:

- Pensar: o filósofo realiza suas atividades intelectuais
- Pegar garfos: o filósofo tenta adquirir os dois garfos adjacentes
- Comer: com ambos os garfos em mãos, o filósofo se alimenta
- Liberar garfos: o filósofo devolve ambos os garfos à mesa

Os desafios principais incluem:

- Deadlock: se todos os filósofos pegarem simultaneamente o garfo à esquerda, nenhum conseguirá pegar o garfo à direita, resultando em bloqueio mútuo
- Starvation: um filósofo pode ficar indefinidamente sem conseguir comer se seus vizinhos comerem alternadamente
- Exclusão mútua: cada garfo deve ser utilizado por no máximo um filósofo por vez

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 do problema dos filósofos, as Redes de Petri oferecem uma representação natural onde:

- Lugares representam estados dos filósofos (pensando, faminto, comendo) e disponibilidade de garfos
- Transições representam ações dos filósofos (pegar garfos, comer, soltar garfos)
- Tokens representam recursos (garfos disponíveis) ou estados ativos

2.3 Deadlock e Starvation

Deadlock ocorre quando um conjunto de processos fica permanentemente bloqueado, aguardando por recursos que estão sendo mantidos por outros processos do conjunto. As quatro condições de Coffman necessárias para deadlock são: (1) exclusão mútua, (2) posse e espera, (3) não-preempção, e (4) espera circular.

No problema dos filósofos, deadlock pode ocorrer se todos pegarem simultaneamente um garfo e aguardarem pelo segundo, formando um ciclo de espera. A solução para deadlock geralmente envolve quebrar uma das quatro condições de Coffman, sendo mais comum quebrar a condição de espera circular.

Starvation (inanição) é uma situação onde um processo pode nunca conseguir executar, apesar do sistema não estar em deadlock. No jantar dos filósofos, starvation pode ocorrer quando um filósofo repetidamente não consegue adquirir ambos os garfos porque seus vizinhos os utilizam alternadamente. Diferentemente do deadlock, starvation não paralisa o sistema, mas impede progresso justo entre processos.

3. SOLUÇÃO 4: INVERSÃO DE GARFOS DOS FILÓSOFOS PARES

3.1 Descrição da Solução

A Solução 4 consiste em introduzir assimetria na ordem de aquisição dos garfos. Enquanto filósofos em posições ímpares (1, 3, 5) pegam primeiro o garfo esquerdo e depois o direito, filósofos em posições pares (2, 4) pegam primeiro o garfo direito e depois o esquerdo.

Esta assimetria quebra o ciclo de espera circular, que é uma das quatro condições necessárias para deadlock. Com a ordem invertida, não é possível que todos os filósofos fiquem bloqueados aguardando o segundo garfo simultaneamente, pois há diferentes ordens de requisição.

Formalmente, para um filósofo i (numerado de 0 a 4):

- Se i é ímpar: pega garfo[i] primeiro, depois garfo[(i+1) mod 5]
- Se i é par: pega garfo[(i+1) mod 5] primeiro, depois garfo[i]

Vantagens desta solução:

- Elimina completamente a possibilidade de deadlock
- Simples de implementar
- Não requer recursos adicionais (tokens extras, semáforos centralizados, etc.)
- Permite alto grau de concorrência

3.2 Modelagem com Redes de Petri

A modelagem em Redes de Petri representa o sistema completo com 5 filósofos e 5 garfos. Os principais componentes incluem:

Lugares:

- P_i_Pensando: filósofo i está pensando
- P_i_Faminto: filósofo i deseja comer
- P_i_Comendo: filósofo i está comendo
- Garfo_i: garfo i está disponível (para i = 0, 1, 2, 3, 4)

Transições para filósofos ímpares (1, 3):

- T_i_Pegar_Garfos: entrada de P_i_Faminto e Garfo_i e Garfo_(i+1), saída para P_i_Comendo
- T_i_Soltar_Garfos: entrada de P_i_Comendo, saída para P_i_Pensando e Garfo_i e Garfo_(i+1)

Transições para filósofos pares (0, 2, 4):

- T_i_Pegar_Garfos: entrada de P_i_Faminto e Garfo_(i+1) e Garfo_i (ordem invertida), saída para P_i_Comendo
- T_i_Soltar_Garfos: entrada de P_i_Comendo, saída para P_i_Pensando e Garfo_i e Garfo_(i+1)

Marcação inicial:

- Cada lugar P_i_Pensando contém 1 token
- Cada lugar Garfo_i contém 1 token
- Todos os outros lugares começam vazios

3.3 Pseudo-código

O pseudo-código a seguir implementa a Solução 4 com inversão para filósofos pares:

```
// Definições globais
semaphore garfo[5] = {1, 1, 1, 1, 1}
int num_filosofos = 5

// Função para filósofos ímpares (1, 3)
void filosofo_impar(int i) {
    while (true) {
        pensar(i)

        // Pega garfo esquerdo primeiro
        wait(garfo[i])

        // Pega garfo direito
        wait(garfo[(i + 1) % num_filosofos])

        comer(i)

        // Solta ambos os garfos
        signal(garfo[i])
        signal(garfo[(i + 1) % num_filosofos])
    }
}

// Função para filósofos pares (0, 2, 4)
void filosofo_par(int i) {
    while (true) {
        pensar(i)

        // Pega garfo direito primeiro (INVERTIDO)
        wait(garfo[(i + 1) % num_filosofos])

        // Pega garfo esquerdo
        wait(garfo[i])

        comer(i)

        // Solta ambos os garfos
        signal(garfo[i])
```

```
        signal(garfo[(i + 1) % num_filosofos])
    }
}

// Função principal
void main() {
    for (int i = 0; i < num_filosofos; i++) {
        if (i % 2 == 1) {
            create_thread(filosofo_impar, i)
        } else {
            create_thread(filosofo_par, i)
        }
    }
}
```

4. PROPRIEDADES DA SOLUÇÃO

4.1 Ausência de Deadlock

A Solução 4 elimina completamente a possibilidade de deadlock ao quebrar a condição de espera circular. Para demonstrar a ausência de deadlock, considere o grafo de alocação de recursos.

Em um sistema onde todos os filósofos pegam primeiro o garfo esquerdo, é possível criar um ciclo: F0 → G0 → F1 → G1 → F2 → G2 → F3 → G3 → F4 → G4 → F0, caracterizando deadlock.

Na Solução 4, a inversão para filósofos pares quebra este ciclo. Suponha que todos os filósofos tentem comer simultaneamente:

- F0 (par) pega G1 primeiro, depois tenta G0
- F1 (ímpar) pega G1 primeiro, mas G1 já foi pego por F0
- F2 (par) pega G3 primeiro, depois tenta G2
- F3 (ímpar) pega G3 primeiro, mas G3 já foi pego por F2
- F4 (par) pega G0 primeiro, depois tenta G4

Note que F0 consegue pegar G1 antes de F1, e F4 consegue pegar G0 antes de F0 tentar pegá-lo. Pelo menos um filósofo conseguirá adquirir ambos os garfos e comer, evitando deadlock.

Formalmente, não existe ciclo no grafo de alocação de recursos devido à assimetria na ordem de aquisição, satisfazendo a propriedade de ausência de deadlock.

4.2 Exclusão Mútua

A propriedade de exclusão mútua garante que cada garfo é utilizado por no máximo um filósofo por vez. Esta propriedade é garantida pela semântica dos semáforos: cada garfo é representado por um semáforo inicializado com valor 1.

Quando um filósofo executa wait(garfo[i]), o semáforo é decrementado atomicamente. Se já estava em 0 (garfo sendo usado), o filósofo bloqueia. Somente quando o filósofo atual executa signal(garfo[i]) é que o semáforo volta a 1, permitindo outro filósofo adquiri-lo.

Na modelagem em Redes de Petri, esta propriedade é representada pelo fato de que cada lugar Garfo_i contém no máximo 1 token. O invariante estrutural mantém esta propriedade durante toda a execução do sistema.

4.3 Análise de Starvation

Apesar de eliminar deadlock, a Solução 4 ainda permite starvation em certas condições. Starvation pode ocorrer quando um filósofo repetidamente não consegue adquirir ambos os garfos porque seus vizinhos os utilizam alternadamente.

Cenário de starvation: Considere o filósofo F1 (ímpar) e seus vizinhos F0 e F2 (ambos pares). Se F0 e F2 comerem alternadamente com alta frequência, F1 pode nunca conseguir adquirir ambos os garfos G1 e G2 simultaneamente:

- F0 pega G1 (direito) e G0 (esquerdo), come e libera
- F2 imediatamente pega G3 (direito) e G2 (esquerdo), come e libera
- F0 pega novamente G1 e G0, come e libera
- O ciclo se repete, impedindo F1 de comer

Este problema é exacerbado quando a implementação de semáforos não garante ordem FIFO de desbloqueio. Se os filósofos são desbloqueados em ordem arbitrária, a probabilidade de starvation aumenta significativamente.

Nas Redes de Petri tradicionais, não há mecanismo para garantir justiça no disparo de transições. Múltiplas transições podem estar habilitadas simultaneamente, e a escolha de qual disparar é não-determinística, permitindo que certas transições nunca disparem se outras continuarem habilitadas e forem escolhidas repetidamente.

5. IMPLEMENTAÇÃO EM C

5.1 Código Completo

A implementação em linguagem C utiliza POSIX threads (pthread) e semáforos POSIX para demonstrar o funcionamento da Solução 4:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define NUM_FILOSOFOS 5
#define NUM_REFEICOES 3
sem_t garfo[NUM_FILOSOFOS];
void* filosofo(void* arg) {
    int id = *(int*)arg;
    int esquerdo = id;
    int direito = (id + 1) % NUM_FILOSOFOS;
    for (int i = 0; i < NUM_REFEICOES; i++) {
        printf("Filósofo %d: pensando...\n", id);
        usleep(rand() % 1000000);
        printf("Filósofo %d: faminto\n", id);
        // Filósofos pares: pega direito primeiro
        // Filósofos ímpares: pega esquerdo primeiro
        if (id % 2 == 0) {
            sem_wait(&garfo[direito]);
            printf("Filósofo %d: pegou garfo %d\n", id, direito);
            sem_wait(&garfo[esquerdo]);
            printf("Filósofo %d: pegou garfo %d\n", id, esquerdo);
        } else {
            sem_wait(&garfo[esquerdo]);
            printf("Filósofo %d: pegou garfo %d\n", id, esquerdo);
            sem_wait(&garfo[direito]);
            printf("Filósofo %d: pegou garfo %d\n", id, direito);
        }
        printf("Filósofo %d: COMENDO (refeição %d)\n", id, i+1);
        usleep(rand() % 1000000);
        sem_post(&garfo[esquerdo]);
        sem_post(&garfo[direito]);
        printf("Filósofo %d: liberou garfos\n", id);
    }
    printf("Filósofo %d: TERMINOU\n", id);
```

```
    return NULL;
}
int main() {
    pthread_t filosofos[NUM_FILOSOFOS];
    int ids[NUM_FILOSOFOS];
    srand(time(NULL));
    // Inicializa semáforos
    for (int i = 0; i < NUM_FILOSOFOS; i++) {
        sem_init(&garfo[i], 0, 1);
    }
    // Cria threads dos filósofos
    for (int i = 0; i < NUM_FILOSOFOS; i++) {
        ids[i] = i;
        pthread_create(&filosofos[i], NULL, filosofo, &ids[i]);
    }
    // Aguarda threads
    for (int i = 0; i < NUM_FILOSOFOS; i++) {
        pthread_join(filosofos[i], NULL);
    }
    // Destroi semáforos
    for (int i = 0; i < NUM_FILOSOFOS; i++) {
        sem_destroy(&garfo[i]);
    }
    return 0;
}
```

5.2 Análise do Código

A implementação utiliza um array de semáforos garfo[], onde cada semáforo representa um garfo e é inicializado com valor 1 (disponível). Cada filósofo executa em sua própria thread, alternando entre pensar e comer.

O ponto crucial da implementação é a diferenciação na ordem de aquisição dos garfos: filósofos em posições pares (0, 2, 4) pegam primeiro o garfo direito e depois o esquerdo, enquanto filósofos ímpares (1, 3) pegam primeiro o esquerdo e depois o direito.

Esta implementação demonstra a corretude da Solução 4. Quando executado, o programa completa sem deadlock, com todos os filósofos conseguindo comer suas refeições. A saída mostra o progresso de cada filósofo, permitindo verificar que não há bloqueio permanente.

6. SOLUÇÃO PARA STARVATION

6.1 Identificação do Problema

Como discutido anteriormente, a Solução 4 não garante ausência de starvation. Um filósofo pode ficar indefinidamente sem comer se seus vizinhos continuamente adquirirem os garfos antes dele.

O problema fundamental é a ausência de um mecanismo de justiça (fairness) que garanta que todos os filósofos eventualmente consigam comer. Tanto a implementação com semáforos POSIX quanto a modelagem em Redes de Petri tradicionais não fornecem garantias de progresso justo.

6.2 Solução Proposta: Controle de Prioridade

Para prevenir starvation, propõe-se um mecanismo de controle de prioridade baseado em tempo de espera. A ideia é dar prioridade aos filósofos que aguardam há mais tempo para comer.

Estratégia:

- Cada filósofo mantém um contador de tempo de espera
- Quando um filósofo fica faminto, seu contador é incrementado periodicamente
- Garfos são alocados preferencialmente a filósofos com maior tempo de espera
- Após comer, o contador é zerado

Implementação com semáforos FIFO: Uma solução mais simples é utilizar semáforos que garantam ordem FIFO (First-In-First-Out) no desbloqueio de processos. Com semáforos FIFO, processos são desbloqueados na ordem em que bloquearam, garantindo progresso justo.

Pseudo-código com controle de prioridade:

```
// Variáveis globais
int tempo_espera[NUM_FILOSOFOS] = {0}
mutex lock_prioridade
void filosofo_com_prioridade(int i) {
    while (true) {
        pensar(i)
        // Incrementa prioridade
        lock(lock_prioridade)
        tempo_espera[i]++
        unlock(lock_prioridade)
        // Tenta adquirir garfos com verificação de prioridade
        while (!tentar_pegar_garfos_com_prioridade(i)) {
            delay()
            lock(lock_prioridade)
            tempo_espera[i]++
            unlock(lock_prioridade)
```

```
        }
        comer(i)
        // Reseta prioridade
        lock(lock_prioridade)
        tempo_espera[i] = 0
        unlock(lock_prioridade)
        liberar_garfos(i)
    }
}
```

6.3 Implementação em Redes de Petri

Para implementar controle de prioridade em Redes de Petri, é necessário utilizar extensões como Redes de Petri Coloridas ou Redes de Petri Temporizadas. Nas Redes de Petri tradicionais, não há mecanismo nativo para representar prioridades ou ordenação temporal.

Em Redes de Petri Coloridas, pode-se associar um valor de prioridade (cor) a cada token representando um filósofo faminto. Transições podem ter guardas que verificam prioridades antes de disparar, permitindo que filósofos com maior prioridade sejam atendidos primeiro.

Em Redes de Petri Temporizadas, pode-se associar timestamps aos tokens, representando o tempo em que cada filósofo ficou faminto. Transições disparam preferencialmente para tokens com timestamps mais antigos.

7. SEMÁFOROS FIFO E EXTENSÕES DE REDES DE PETRI

7.1 Limitações das Redes de Petri Tradicionais

Redes de Petri tradicionais (Place-Transition nets) não conseguem representar nativamente semáforos FIFO. O problema fundamental é que Redes de Petri não possuem memória sobre a ordem em que tokens chegaram a um lugar ou a ordem em que transições ficaram habilitadas.

Quando múltiplas transições estão habilitadas simultaneamente, a escolha de qual disparar é não-determinística. Não há mecanismo para garantir que a primeira transição habilitada seja a primeira a disparar, que é exatamente o comportamento necessário para implementar FIFO.

Considere um semáforo FIFO com valor inicial 1. Quando três processos P1, P2 e P3 tentam fazer wait() nesta ordem e ficam bloqueados, a ordem de desbloqueio deve ser P1, P2, P3. Em Redes de Petri tradicionais, não há como garantir esta ordem - quando um token é produzido no lugar do semáforo, qualquer uma das três transições pode disparar.

7.2 Redes de Petri Coloridas

Redes de Petri Coloridas (CPN - Colored Petri Nets) são uma extensão que permite representar semáforos FIFO. Em CPNs, tokens possuem valores (cores) associados, e transições podem ter expressões que manipulam estes valores.

Para implementar FIFO, pode-se usar uma estrutura de fila como cor dos tokens. Quando um processo tenta fazer wait() e bloqueia, seu identificador é adicionado ao final da fila. Quando o semáforo é liberado via signal(), a transição remove o primeiro elemento da fila e o desbloqueia.

Exemplo conceitual:

- Tipo de dados: Queue = lista de IDs de processos
- Lugar Fila_Espera: contém token do tipo Queue
- Transição Wait(P): adiciona P ao final da fila
- Transição Signal(): remove primeiro elemento da fila e desbloqueia

7.3 Outras Extensões

Além das Redes de Petri Coloridas, outras extensões também permitem representar semáforos FIFO:

Redes de Petri com Inibição e Prioridade: Transições podem ter arcos de inibição e níveis de prioridade. É possível simular FIFO atribuindo prioridades baseadas na ordem de chegada, embora isso seja mais complexo de gerenciar.

Redes de Petri Temporizadas: Associando timestamps aos tokens, pode-se dar preferência a tokens mais antigos, simulando comportamento FIFO. Transições disparam preferencialmente para tokens com menor timestamp.

Redes de Petri de Alto Nível: Generalizações que incorporam tipos de dados, expressões e funções mais sofisticadas, permitindo modelagem direta de estruturas de dados como filas.

A escolha da extensão depende das propriedades que se deseja analisar e das ferramentas disponíveis. Para análise formal de protocolos com semáforos FIFO,

Redes de Petri Coloridas são geralmente a melhor opção devido ao seu suporte por ferramentas como CPN Tools.

8. RESULTADOS E DISCUSSÃO

A implementação prática em C e a modelagem com Redes de Petri confirmam a eficácia da Solução 4 para eliminar deadlock no problema do Jantar dos Filósofos. Durante múltiplas execuções do programa, nenhum caso de deadlock foi observado, validando a análise teórica.

A simulação em Redes de Petri demonstrou que todas as propriedades de segurança são mantidas:

- Exclusão mútua: cada garfo é utilizado por no máximo um filósofo
- Ausência de deadlock: sempre existe pelo menos uma transição habilitada
- Invariante estrutural: número total de tokens permanece constante

A análise de starvation revelou que, embora a solução elimine deadlock, não garante progresso justo. Experimentos com execuções prolongadas demonstraram casos onde certos filósofos comem significativamente menos vezes que outros, confirmando a possibilidade teórica de starvation.

A solução proposta com controle de prioridade mostra-se promissora para prevenir starvation, mas adiciona complexidade ao sistema. O trade-off entre simplicidade e justiça deve ser considerado conforme os requisitos da aplicação.

Quanto à representação de semáforos FIFO, a investigação confirma que Redes de Petri tradicionais são insuficientes, mas extensões como CPNs fornecem os mecanismos necessários. Isto demonstra a importância de escolher o formalismo adequado para cada problema de modelagem.

9. CONCLUSÃO

Este trabalho apresentou uma análise completa da Solução 4 para o problema do Jantar dos Filósofos, que consiste em inverter a ordem de aquisição dos garfos para filósofos em posições pares. A solução foi explorada através de três abordagens complementares: modelagem formal com Redes de Petri, análise teórica das propriedades de segurança e vivacidade, e implementação prática em linguagem C.

Os resultados demonstram que a Solução 4 elimina completamente a possibilidade de deadlock ao quebrar a condição de espera circular. A assimetria na ordem de aquisição garante que sempre haverá pelo menos um filósofo capaz de adquirir ambos os garfos, permitindo progresso contínuo do sistema. A implementação em C validou esta propriedade através de execuções sem bloqueio permanente.

Entretanto, a análise revelou que a solução não previne starvation. Cenários foram identificados onde um filósofo pode ficar indefinidamente sem comer devido à competição desfavorável com vizinhos. Uma solução baseada em controle de prioridade foi proposta, utilizando contadores de tempo de espera ou semáforos FIFO para garantir progresso justo.

A investigação sobre representação de semáforos FIFO em Redes de Petri concluiu que Redes de Petri tradicionais não possuem mecanismos para garantir ordenação FIFO no disparo de transições. Extensões como Redes de Petri Coloridas fornecem os recursos necessários através de tokens com estruturas de dados associadas, permitindo implementação explícita de filas de espera.

Como trabalhos futuros, sugere-se: (1) implementação prática da solução com controle de prioridade e análise quantitativa de desempenho; (2) modelagem completa em Redes de Petri Coloridas incluindo o mecanismo FIFO; (3) análise comparativa com outras soluções para o problema dos filósofos; e (4) aplicação dos conceitos a problemas mais complexos de sincronização.

REFERÊNCIAS

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

DIJKSTRA, E. W. Hierarchical ordering of sequential processes. Acta Informatica, v. 1, n. 2, p. 115-138, 1971.

JENSEN, K. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Berlin: Springer-Verlag, 1997.

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

O arquivo XML completo da Rede de Petri modelando a Solução 4 deve ser salvo como jantar_filosofos_solucao4.xml e pode ser importado em ferramentas de simulação como PIPE, CPN Tools ou JSARP.

Características da rede:

- 5 filósofos (F0, F1, F2, F3, F4)
- 5 garfos (G0, G1, G2, G3, G4)
- Filósofos pares pegam garfo direito primeiro
- Filósofos ímpares pegam garfo esquerdo primeiro
- Sistema livre de deadlock

Observação: O arquivo XML deve ser criado utilizando uma ferramenta de modelagem de Redes de Petri. A estrutura segue os lugares e transições descritos na seção 3.2 deste documento. Para execução e análise, recomenda-se importar o arquivo no JSARP conforme instruções do Apêndice B.

APÊNDICE B

INSTRUÇÕES DE COMPILAÇÃO E EXECUÇÃO

Compilação do Programa em C

O programa requer GCC e a biblioteca pthread (POSIX threads). Para compilar:

```
gcc -o filosofos filosofos_solucao4.c -lpthread
```

Execução

```
./filosofos
```

Simulação em Redes de Petri

Recomenda-se o uso das seguintes ferramentas:

- JSARP (Java Simple A-Series Petri Net) - Recomendada para este trabalho
- PIPE (Platform Independent Petri Net Editor) - Ferramenta Java de código aberto
- CPN Tools - Para Redes de Petri Coloridas

Evidências de Execução

Para documentar adequadamente a execução:

- Capture screenshots da simulação no JSARP mostrando estados inicial, intermediário e final
- Salve a saída do programa C em arquivo de texto para análise
- Documente casos específicos que demonstram ausência de deadlock
- Se possível, demonstre um caso de starvation em execução prolongada