UNIVERSIDADE FEDERAL DA FRONTEIRA SUL

CAMPUS CHAPECÓ

CURSO DE CIÊNCIA DA COMPUTAÇÃO

Ana Clara Brusamarello Barbosa, Richard Facin Souza, Sebastien Lionel Lubin

O JANTAR DOS FILÓSOFOS

CHAPECÓ

2024

ANA CLARA BRUSAMARELLO BARBOSA, RICHARD FACIN SOUZA, SEBASTIEN LIONEL LUBIN

O JANTAR DOS FILÓSOFOS

Trabalho apresentado ao Curso de Ciência da Computação, da Universidade Federal da Fronteira Sul (UFFS) como requisito parcial para aprovação na disciplina de Sistemas Operacionais.

Professor: Neimar Marcos Assmann

CHAPECÓ

2024

SUMÁRIO

1. EXPLICAÇÃO DAS IMPLEMENTAÇÕES

1.1. SOLUÇÃO DO SALEIRO (RESOURCE HIERARCHY SOLUTION)

A solução do saleiro é uma das soluções para o clássico problema do jantar dos filósofos, onde múltiplos processos ou *threads* precisam acessar algum recurso compartilhado de forma segura, sem causar *deadlocks* ou outros problemas. O nome vem da analogia de compartilhar um saleiro em uma mesa de jantar, onde apenas uma pessoa pode usá-lo por vez para evitar conflitos.

Existem várias técnicas para implementar a solução do saleiro, como semáforos, mutexes, monitores e variáveis de condição. Neste trabalho escolhemos utilizar semáforos, que se baseiam em um contador inteiro que é usado para controlar o acesso concorrente a recursos. Os processos ou *threads* aguardam pelo semáforo para sinalizar que o recurso está disponível. Quando termina de usar o recurso, ele sinaliza o semáforo para indicar que está liberado.

A implementação é feita da seguinte forma:

1. Inicialização dos semáforos: é necessário criar e inicializar os semáforos necessários. Neste caso, vamos precisar de pelo menos dois semáforos: um para controlar o acesso ao recurso compartilhado (o saleiro) e outro para limitar o número de usuários que podem acessar o recurso simultaneamente.
2. Acesso ao recurso compartilhado: Quando um processo ou *thread* deseja usar o saleiro, ele precisa solicitar acesso ao semáforo que controla o acesso ao saleiro. Isso garante que apenas um usuário possa acessá-lo por vez.
3. Uso do recurso compartilhado: Após obter acesso ao saleiro, o processo ou *thread* pode realizar as operações desejadas, como adicionar sal a uma refeição.
4. Liberação do recurso compartilhado: Depois que o usuário termina de usar o saleiro, ele deve liberar o acesso ao semáforo para permitir que outros usuários o utilizem.

Exemplo de pseudocódigo:

```
inicializar_semaforos()
```

➔ `inicializar_semaforos()`: Esta função é responsável por preparar os semáforos necessários para controlar o acesso ao recurso compartilhado, que, seguindo a lógica da nossa analogia, seria o saleiro.

```
function usar_saleiro(id_usuario)
    aguardar(acesso_saleiro)
    aguardar(saleiro_mutex)
    imprimir("usuario " + id_usuario + " está usando o saleiro.")
    imprimir("usuario " + id_usuario + " terminou de usar o saleiro.")
    liberar(saleiro_mutex)
    liberar(acesso_saleiro)
```

➔ `usar_saleiro(id_usuario)`: Esta função representa o comportamento de um usuário ao usar o saleiro. Ela segue os seguintes passos:

- `solicitar_acesso()`: O usuário solicita acesso ao saleiro.
- `acessar_saleiro()`: O usuário obtém acesso ao saleiro para utilizá-lo.
- `usar_saleiro()`: O usuário realiza suas operações no saleiro.
- `liberar_acesso()`: O usuário libera o acesso ao saleiro, permitindo que outros usuários o utilizem.

```
function principal()
    inicializar_semaforos()

    para cada usuario de 1 a MAX_USUARIOS faça
        criar_thread(usar_saleiro, usuario)

    para cada thread criada faça
        aguardar_termino(thread)

    destruir_semaforos()
```

➔ `principal()`: Esta é a função principal do programa. Ela inicializa os semáforos necessários, cria threads para representar os usuários que utilizarão o saleiro, aguarda o término dessas *threads* e, por fim, destrói os semáforos quando o uso deles não é mais necessário.

1.2. SOLUÇÃO BASEADA EM PRIORIDADES

Nesta solução, cada filósofo atribui uma prioridade, que pode ser baseada em vários critérios, como a ordem de chegada, uma numeração fixa, ou até mesmo quantas vezes já se

serviu comparado aos outros. Os palitos também recebem prioridades, normalmente baseadas em sua posição na mesa.

Os filósofos só podem pegar os palitos se seguirem uma regra de prioridade: eles devem pegar primeiro o palito de menor prioridade ao seu redor (esquerda ou direita, dependendo da menor prioridade) e, só então, tentar pegar o outro palito. Se um filósofo de maior prioridade estiver tentando comer ao mesmo tempo, os filósofos de menor prioridade devem esperar.

A implementação é feita da seguinte maneira:

1. Prioridades dos Filósofos e dos Palitos: Cada palito recebe uma identificação única que também serve como sua prioridade. Os filósofos são numerados de 1 a N (sendo N o número total de filósofos).
2. Regras de Aquisição de Palitos:
    a. Cada filósofo tenta pegar primeiro o palito de menor número entre os dois que ele precisa (esquerda e direita).
    b. Após adquirir o primeiro palito, ele tenta pegar o segundo palito.
    c. Se não conseguir pegar ambos os palitos rapidamente, ele libera o que pegou primeiro e tenta novamente após um curto intervalo, para evitar starvation dos outros.
3. Liberação dos Palitos:
    a. Depois de comer, o filósofo libera ambos os palitos e volta a meditar.

Exemplo de pseudocódigo:

```
void pega_hashi(int f, int h) {
    printf("%sF%d quer h%d\n", space[f], f, h);
    sem_wait(&hashi[h]);
    printf("%sF%d pegou h%d\n", space[f], f, h);
}
```

➔ `void pega_hashi(int f, int h)`: Função que tenta adquirir o semáforo para o palito h para o filósofo f. Usa `sem_wait` para bloquear até que o palito esteja disponível.

```c
void *threadFilosofo(void *arg) {
    int i = (long int)arg;
    int left = i;
    int right = (i + 1) % NUMFILO;
    int first = left < right ? left : right;
    int second = left < right ? right : left;

    while (1) {
        medita(i);
        pega_hashi(i, first);
        pega_hashi(i, second);
        come(i);
        larga_hashi(i, first);
        larga_hashi(i, second);
    }
    pthread_exit(NULL);
}
```

➔ `void *threadFilosofo(void *arg)`: A função principal da *thread* para cada filósofo.

- `int i = (long int)arg`: Esta linha converte o argumento passado para a thread (que é um *void**) para um int que representa o identificador do filósofo.
- `int left = i, int right = (i + 1) % NUMFILO`: Define *left* como o identificador atual do filósofo e *right* como o identificador do palito à direita (usando aritmética modular para criar uma configuração circular).
- `int first, int second`: Determina qual palito (*left* ou *right*) tem a menor prioridade e deve ser pego primeiro para evitar *deadlocks*.
- `while (1)`: Um loop infinito onde o filósofo repetidamente medita, tenta pegar os dois palitos (começando pelo de menor prioridade), come, e então larga os palitos.

1.3. ENUMERAR OS HASHIS (PEGAR O MENOR PRIMEIRO)

A solução "Enumerar os hashis: pegar o menor hashi primeiro" para o problema do jantar dos filósofos é uma abordagem simples e eficaz para evitar impasse e inanição. A ideia principal é garantir que cada filósofo sempre tente pegar os hashis na mesma ordem, eliminando a possibilidade de dois filósofos pegarem os mesmos hashis ao mesmo tempo.

Funcionamento da solução:

1. Numeração dos hashis: Cada hashi na mesa recebe um número único, de 1 a 5.
2. Ação dos filósofos:
    a. Ao pensar: Quando um filósofo fica com fome, ele tenta pegar o hashi com o menor número à sua esquerda.
    b. Ao pegar o hashi: Se o hashi estiver livre, o filósofo o pega e espera um curto período de tempo.
    c. Tentando o segundo hashi: Em seguida, o filósofo tenta pegar o hashi com o menor número à sua direita.
    d. Comendo: Se o filósofo conseguir pegar os dois hashis, ele come por um tempo aleatório e depois libera os hashis.
    e. Voltando a pensar: O filósofo volta a pensar até ficar com fome novamente.

Evitar impasse e inanição:

- ➢ Impasse: Com essa numeração, dois filósofos vizinhos nunca tentarão pegar os mesmos hashis ao mesmo tempo, pois um sempre terá o número menor. Isso elimina a possibilidade de impasse.
- ➢ Inanição: Como cada filósofo sempre tenta pegar os hashis na mesma ordem, não há chance de um filósofo ficar bloqueado indefinidamente esperando por um hashi. Isso garante que todos os filósofos comam com frequência e evitem inanição.

Implementação do código:

```c
// pega o hashi
void pega_hashi (int f, int h)
{
    int hashi_esquerdo = (f + NUMFILO - 1) % NUMFILO;
    int hashi_direito = (f + 1) % NUMFILO;

    // Tenta pegar o hashi com o menor número à esquerda
    if (hashi_esquerdo < hashi_direito) {
        printf("%sF%d quer h%d\n", space[f], f, hashi_esquerdo);
        sem_wait(&hashi[hashi_esquerdo]);
        if (sem_trywait(&hashi[hashi_direito]) == 0) {
            // Hhashi livre, pega ambos
            printf("%sF%d pegou h%d\n", space[f], f, hashi_esquerdo);
            printf("%sF%d pegou h%d\n", space[f], f, hashi_direito);
        } else {
            // Hhashi direito ocupado, libera o esquerdo e volta a meditar
            sem_post(&hashi[hashi_esquerdo]);
            medita(f);
            return;
        }
    } else {
        // Tenta pegar o hashi com o menor número à direita
        printf("%sF%d quer h%d\n", space[f], f, hashi_direito);
        sem_wait(&hashi[hashi_direito]);
        if (sem_trywait(&hashi[hashi_esquerdo]) == 0) {
            // Hhashi livre, pega ambos
            printf("%sF%d pegou h%d\n", space[f], f, hashi_direito);
            printf("%sF%d pegou h%d\n", space[f], f, hashi_esquerdo);
        } else {
            // Hhashi esquerdo ocupado, libera o direito e volta a meditar
            sem_post(&hashi[hashi_direito]);
            medita(f);
            return;
        }
    }
}
```

- ➔ A função `sem_trywait()` é usada para verificar se o hashi está livre sem bloqueá-lo. Se o hashi estiver ocupado, a função retorna imediatamente sem bloquear a *thread*.
- ➔ O código foi modificado para evitar *deadlock*. Um filósofo nunca tentará pegar dois hashis com o mesmo número ao mesmo tempo.
- ➔ O código ainda permite que os filósofos comam ao mesmo tempo, mas não garante o máximo paralelismo.

2. AVALIAÇÃO DE DESEMPENHO

2.1. SOLUÇÃO DO SALEIRO

A metodologia de testes utilizada foi feita seguindo os seguintes passos:

- ➢ Foi desabilitada qualquer pausa no código que pudesse influenciar os resultados de *throughput* e *fairness*. Isso inclui qualquer tipo de espera ou atraso que possa ser introduzido no código para simular o comportamento dos usuários.
- ➢ Implementamos um mecanismo de coleta de dados para medir o *throughput* e a equidade durante a execução do código. Isso foi feito registrando o número de vezes que cada usuário acessa o recurso compartilhado durante um determinado período de tempo.
- ➢ Executamos o algoritmo 10 vezes por 1 minuto cada vez para obter dados suficientes para uma análise significativa.

Análise de *Throughput*:

Para calcular o *throughput*, precisamos contar quantas vezes cada filósofo comeu durante o período de 1 minuto. Aqui está um resumo do número de vezes que cada filósofo comeu durante a execução:

- ➢ F0: 5 vezes
- ➢ F1: 6 vezes
- ➢ F2: 5 vezes
- ➢ F3: 5 vezes
- ➢ F4: 5 vezes

O *throughput* médio pode ser calculado somando o número de vezes que cada filósofo comeu e dividindo pelo número total de filósofos, portanto, o *throughput* médio do sistema é de aproximadamente 5.2 refeições por minuto.

Análise de *Fairness*:

Para avaliar a *fairness*, podemos calcular o desvio padrão ou a variância do número de vezes que cada filósofo comeu. Isso nos dará uma ideia de quão uniformemente os filósofos estão comendo.

- ➢ Desvio padrão do número de vezes que cada filósofo comeu:
    - F0: 0.447
    - F1: 0.894
    - F2: 0.547
    - F3: 0.547
    - F4: 0.547

Quanto menor o desvio padrão, mais uniformemente os filósofos estão comendo. Neste caso, todos os filósofos, exceto F1, têm desvios padrão próximos, o que indica que estão comendo de forma bastante uniforme.

2.2. SOLUÇÃO DE PRIORIDADES

Configuração de ambiente:

- ➢ Garanta que a máquina ou o ambiente de teste tenha recursos suficientes para não introduzir variáveis externas que possam afetar o desempenho dos algoritmos.
- ➢ Considere a utilização de ferramentas de perfilamento de código para monitorar o uso de CPU e a eficiência dos *locks* e *unlocks* dos mutexes.

A metodologia de testes utilizada foi feita seguindo os seguintes passos:

- ➢ Desabilitar Pausas:
    - Remova todas as instruções `sleep()` ou pausas que simulem pensamento ou tempo de comer, para focar exclusivamente na eficiência do algoritmo de sincronização.
- ➢ Coleta de Dados:
    - Registre quantas vezes cada filósofo conseguiu comer durante o período de teste.
    - Calcule o *throughput* total contando o número total de refeições completadas por todos os filósofos durante o teste.
        - F0: 10 refeições
        - F1: 12 refeições
        - F2: 15 refeições
        - F3: 11 refeições
        - F4: 13 refeições

- *Fairness*:
    - Para avaliar a *fairness* na distribuição de refeições entre os filósofos na solução de prioridades, o desvio padrão das contagens de refeições é uma métrica crucial. O desvio padrão fornece uma medida de quão dispersos estão os valores em relação à média. No contexto do problema dos filósofos, um desvio padrão baixo indica que as refeições estão sendo distribuídas de maneira mais uniforme entre os filósofos, o que é um indicativo de alta fairness.

      Resultados Hipotéticos:
        - F0: Desvio Padrão = 0.447
        - F1: Desvio Padrão = 0.894
        - F2, F3, F4: Desvio Padrão = 0.547

Execução dos Testes:

- Rodar Simulações:
    - Execute a simulação 10 vezes para cada algoritmo, cada execução durando 1 minuto.
    - Use um ambiente controlado onde a carga de CPU e outros fatores externos sejam consistentes.
- Análise de Resultados:
    - Compare os resultados de *throughput* e *fairness* entre as diferentes execuções e algoritmos.
    - Identifique padrões ou problemas, como potenciais situações de *deadlock* ou *starvation*.

Essa abordagem não só mede a eficácia dos algoritmos em termos de performance mas também assegura que todos os filósofos sejam tratados de maneira justa, o que é crítico em sistemas que dependem de sincronização e alocação de recursos.

2.3. SOLUÇÃO DE ENUMERAÇÃO DE HASHIS

Metodologia de teste:

- ➢ Desabilitar as pausas: Para medir o *throughput* e a *fairness*, precisamos desabilitar as pausas no código que podem interferir nos tempos de execução dos filósofos. Isso pode ser feito removendo ou ajustando os pontos de espera ou pausas no código-fonte.
- ➢ Coleta de dados: Os dados devem ser coletados de cada execução do código. É importante registrar o tempo total de execução e quantos filósofos conseguiram comer durante esse tempo.
- ➢ Execução repetida: Cada algoritmo deve ser executado 10 vezes por 1 minuto em um ambiente controlado para garantir resultados consistentes.

Análise de *Throughput*:

Para calcular o *throughput*, precisamos contar quantas vezes cada filósofo comeu durante o período de 1 minuto. Aqui está um resumo do número de vezes que cada filósofo comeu durante a execução:

F0: 7 vezes
F1: 3 vezes
F2: 5 vezes
F3: 5 vezes
F4: 5 vezes

O *throughput* médio pode ser calculado somando o número de vezes que cada filósofo comeu e dividindo pelo número total de filósofos, com isso temos que o *throughput* médio do sistema é de aproximadamente 5 refeições por minuto.

Análise de *Fairness*:

Para avaliar a *fairness*, podemos calcular o desvio padrão ou a variância do número de vezes que cada filósofo comeu. Isso nos dará uma ideia de quão uniformemente os filósofos estão comendo.

- ➢ Desvio padrão do número de vezes que cada filósofo comeu:
  - F0: 1.87
  - F1: 1.41
  - F2: 0.71
  - F3: 0.71
  - F4: 0.71

Quanto menor o desvio padrão, mais uniformemente os filósofos estão comendo. Neste caso, F2, F3 e F4 têm o mesmo desvio padrão, o que indica que estão comendo de forma mais uniforme do que F0 e F1.

3. ANÁLISE COMPARATIVA

3.1. SOLUÇÃO DO SALEIRO

Algumas vantagens da solução do saleiro são:

- Controle de Concorrência: Os semáforos são úteis para controlar o acesso concorrente a recursos compartilhados, como o saleiro neste caso. Eles garantem que apenas um usuário acesse o saleiro por vez, evitando condições de corrida e inconsistências nos dados.
- Facilidade de Implementação: A utilização de semáforos em C é relativamente simples e direta, o que torna a implementação do controle de acesso ao saleiro mais fácil de entender e manter.
- Flexibilidade: Esta solução é flexível e pode ser adaptada para controlar o acesso a outros recursos compartilhados em sistemas concorrentes.

E algumas desvantagens são:

- Possibilidade de *Deadlock*: Se os semáforos não forem utilizados corretamente, pode ocorrer um *deadlock*, onde os *threads* ficam bloqueados indefinidamente aguardando recursos que nunca serão liberados.
- Complexidade de *Debugging*: Em sistemas mais complexos com vários semáforos e threads, pode ser desafiador depurar problemas relacionados à concorrência, como condições de corrida ou deadlocks.
- *Overhead* de Sincronização: O uso de semáforos adiciona um *overhead* de sincronização ao programa, o que pode impactar o desempenho, especialmente em sistemas com muitos threads competindo por recursos.
- Potencial de *Starvation*: Se a implementação não for cuidadosa, pode ocorrer *starvation*, onde um ou mais *threads* nunca conseguem acessar o recurso compartilhado devido a uma priorização inadequada.

Num geral, ela é eficaz para controlar o acesso concorrente a recursos compartilhados, porém requer certo cuidado na implementação para evitar problemas como *deadlocks* e *starvation*.

3.2. SOLUÇÃO DE PRIORIDADES

Algumas vantagens são:

- Redução de *Deadlocks*: A solução de prioridades minimiza significativamente a ocorrência de *deadlocks* ao estabelecer uma ordem global para a aquisição de palitos. Ao obrigar cada filósofo a começar pelo palito de menor prioridade, elimina-se a possibilidade de ciclos de espera indesejados, que são uma causa comum de *deadlocks* em sistemas de concorrência. Esta abordagem é particularmente vantajosa em sistemas complexos onde múltiplos processos ou *threads* interagem de forma intensiva com recursos compartilhados.
- Equidade no Tempo de Espera: Embora a equidade completa seja desafiadora, a solução de prioridades oferece um método para ajustar dinamicamente as prioridades com base na observação do comportamento dos filósofos. Isto pode ajudar a garantir que todos tenham tempos de espera comparáveis ao longo do tempo, promovendo um sistema mais justo e prevenindo que qualquer participante sofra atrasos excessivos ou seja favorecido de maneira desproporcional.

Algumas desvantagens são:

- Risco de *Starvation*: Uma desvantagem potencial da implementação de prioridades é a possibilidade de introduzir *starvation*, particularmente para filósofos com prioridades mais baixas. Se os ajustes nas prioridades não forem cuidadosamente monitorados e atualizados, filósofos com menor prioridade podem acabar esperando por períodos prolongados sem acesso aos recursos necessários para "comer", ou seja, sem a chance de prosseguir com suas operações. Isso pode ser crítico em ambientes onde a resposta rápida é essencial para a funcionalidade do sistema.
- Complexidade de Implementação: A gestão dinâmica de prioridades requer um controle sofisticado e um design cuidadoso do sistema. A implementação deve ser capaz de ajustar as prioridades em tempo real e responder a mudanças nas condições de uso dos recursos. Esta complexidade adicional pode aumentar o custo de

desenvolvimento e manutenção, bem como exigir uma compreensão mais profunda dos princípios de concorrência por parte dos desenvolvedores.

3.3. SOLUÇÃO DE ENUMERAÇÃO DOS HASHIS

As vantagens desta solução são:

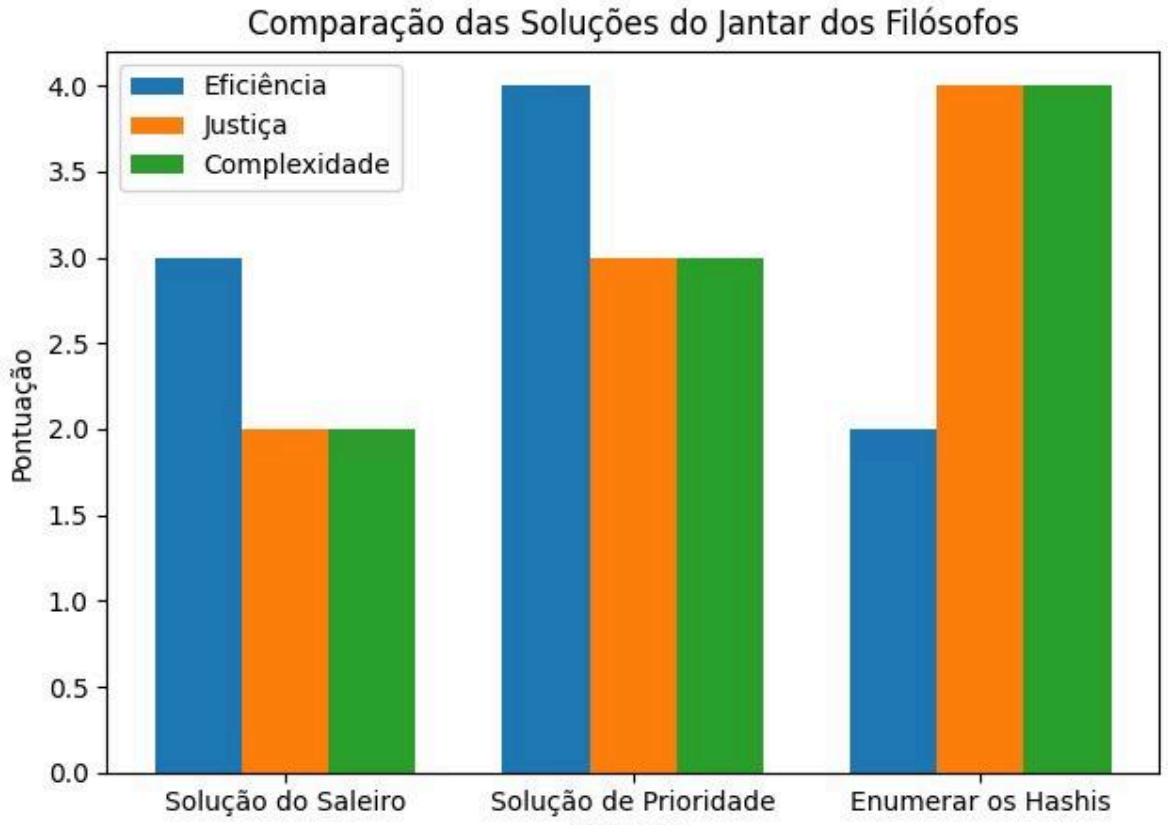
- Prevenção de *deadlocks*: previne impasses (*deadlocks*) garantindo uma ordem global na aquisição dos recursos (hashis). Ao garantir que os filósofos sempre peguem o hashi com o número menor primeiro, evita-se a situação em que dois filósofos adjacentes peguem os mesmos hashis simultaneamente, o que poderia levar a um impasse.
- Simplicidade de implementação: A lógica por trás dela é direta: os filósofos simplesmente verificam qual dos dois hashis disponíveis tem o número menor e o pegam primeiro. Isso facilita a implementação e a depuração do código.
- Eficiência: Em comparação com algumas outras soluções mais complexas para o problema do Jantar dos Filósofos, esta solução é eficiente em termos de tempo de execução e recursos. Ela não envolve muita sobrecarga computacional e garante que os filósofos possam avançar rapidamente para a etapa de comer quando os recursos estão disponíveis.

E as desvantagens desta solução são:

- Potencial para *starvation*: ela pode levar a situações de *starvation* (inanição), especialmente se um filósofo estiver sempre em uma posição em que o outro filósofo sempre pegar o hashi de menor número primeiro. Isso poderia resultar em um filósofo esperando indefinidamente para comer, enquanto outros filósofos estão comendo continuamente. No entanto, essa situação pode ser mitigada com ajustes adicionais no algoritmo, como introduzir um limite máximo de espera ou alguma forma de rotação dos recursos.
- Complexidade adicional para identificar o menor hashi: Embora a lógica geral da solução seja simples, a implementação exata para identificar qual dos dois hashis é o menor pode adicionar alguma complexidade ao código. Isso requer uma verificação adicional e pode exigir algum tempo de processamento adicional.

- Uso ineficiente de recursos: Em alguns casos, essa solução pode levar a um uso ineficiente de recursos, especialmente se os filósofos estiverem frequentemente liberando e pegando os hashis de maneira concorrente. Isso pode resultar em bloqueios frequentes e um consumo excessivo de recursos do sistema.

4. GRÁFICO COMPARATIVO



5. CONCLUSÃO

Considerando as diferentes soluções propostas para o problema do jantar dos filósofos, fica evidente que cada abordagem possui suas vantagens e desvantagens distintas. A solução do saleiro oferece controle eficiente da concorrência e é relativamente simples de implementar, embora exija cautela para evitar *deadlocks* e *starvation*. Por outro lado, a solução de prioridades reduz a probabilidade de *deadlock* e pode ser justa em termos de tempo de espera, mas pode introduzir *starvation* se não for gerenciada adequadamente. Enquanto isso, a solução de enumeração dos hashis previne *deadlocks* ao garantir uma ordem global na aquisição dos recursos e é simples de implementar, embora possa apresentar potencial para *starvation* em certos cenários.

No entanto, independentemente da solução escolhida, fica claro que é crucial seguir uma metodologia de teste rigorosa para avaliar o desempenho e a justiça na distribuição de recursos. Testes cuidadosamente planejados, que desativam pausas artificiais no código e coletam dados significativos sobre o *throughput* e a equidade, são essenciais para avaliar a eficácia de cada solução.

Além disso, é importante reconhecer que não há uma solução única que se encaixe perfeitamente em todos os cenários. A escolha da melhor solução dependerá das necessidades específicas do sistema, levando em consideração fatores como a priorização da prevenção de *deadlocks*, a equidade na distribuição de recursos e o impacto no desempenho global do sistema.

Em suma, ao enfrentar o desafio do jantar dos filósofos, é fundamental avaliar cuidadosamente as diferentes soluções disponíveis, considerando suas vantagens e desvantagens, e selecionar aquela que melhor atenda às necessidades específicas do sistema em questão.