# Relatório do Trabalho 1 de Sistemas Operacionais

Dhruv Babani
Eduardo Barcellos
Eduardo Felber
Tomaz Bettio
Pontifícia Universidade Católica do Rio Grande do Sul — PUCRS

10 de abril de 2024

# 1 Introdução

Neste trabalho, iremos descrever como resolvemos dois problemas muito comuns quando falamos de threads e a sincronização deles. Como havia dito, os nomes dos dois problemas são Produtores e Consumidores e o Jantar dos Canibais.Por tanto, primeiro vamos mostrar a implementação do mutex, e depois as soluções detalhadamente.

# 2 Implementação do Mutex(Semáforo Binário)

## 2.1 Produtor e Consumidores -> Algoritmo de Peterson

Com base na descrição do enunciado do trabalho, não podíamos utilizar os metodos de lock e unlock da biblioteca pthread, com o objetivo de entrar e sair de uma secção crítica. Então, tivemos que implementá-los.

1. MutexLock(int self)
    - flag[self] = 1: Aqui, o processo marca sua intenção de entrar na seção crítica, definindo flag[self] como 1;
    - turn = 1 - self: O turn é uma variável compartilhada que controla qual processo pode entrar na seção crítica. Se o processo atual (identificado por self) não é o próximo na fila (ou seja, turn), ele espera até que seja seu turno;
    - while (flag[1 - self] == 1 turn == 1 - self) : Este é o loop de espera. Enquanto o outro processo (1 - self) estiver na seção crítica (flag[1 - self] == 1) e é seu turno (turn == 1 - self), o processo atual espera até que seja seguro entrar na seção crítica. Isso é conhecido como uma espera ativa, onde o processo verifica constantemente até que a condição de entrada segura seja atendida.
2. MutexUnlock(int self)
    - flag[self] = 0: Aqui, o processo sinaliza que ele não está mais na seção crítica, definindo flag[self] como 0.

Agora com os metodos para entrar e sair de uma secção crítica já estão implementados, podemos resolver o problema citado anteriormente.

## 2.2 Jantar dos Canibais -> Algoritmo de Lamport

Já para o segundo o problema, implementamos o algoritmo de Lamport, para contrlar a secção crítica atráves do Semaforo Binário. Aqui está o detalhamento da implementação:

1. Lock(processo)
    - O método lock é responsável por permitir que um thread adquira o bloqueio do recurso compartilhado;
    - Primeiro, atribui ao thread atual um número de sequência maior que qualquer outro thread;
    - Em seguida, verifica se é a vez do thread atual de acessar o recurso, com base nos números de sequência dos outros threads;
    - e não for sua vez, entra em um loop de espera até que seja.
2. Unlock(processo)
    - O método unlock é responsável por liberar o bloqueio do recurso compartilhado;
    - Simplesmente atribui 0 ao número de sequência do thread atual, indicando que o recurso está livre para ser adquirido por outros threads.

# 3 Primeiro Problema: Produto e Consumidor

Para resolver o problema, separamos a logica em algumas secções, tais como, as variaveis globais, a função Produtor e Consumidor(Operações responsáveis por cada um) e função main, onde irá disparar as threads Consumidores e Produtores, e apresenterá todo o fluxo do programa.Então, aqui está a solucão com as todas as secções correspodentes:

1. Variáveis Globais:
    - Definimos duas constantes para determinar se o buffer está vazio ou cheio;
    - Declaramos uma variavel buffer, que seria responsavel para receber o item a ser inserido;
    - Declaramos uma variavel estado, que recebe de inicio que o buffer está vazio.
2. Função Produtor:
    - Primeiramente, um loop é inicializado, para definir o numero de itens a ser produzidos, que na qual seriam apenas 10;
    - Dentro do loop,o produtor produz um item, calculando item = i + (id * 1000). Isso garante que cada produtor produza itens com valores distintos;
    - Após produzir um item, o código entra em um loop do-while, que continua até que seja possível inserir o item no buffer. Isso é feito verificando o estado do buffer.Dentro do loop do-while, o produtor tenta obter um bloqueio de mutex (um mecanismo de sincronização) usando a função mutexlock(id). Isso é feito para garantir que apenas um produtor manipule o buffer por vez;
    - Após obter o bloqueio do mutex, o código verifica se o estado do buffer é BUFFERCHEIO. Se estiver cheio, o produtor deve aguardar, então a variável aguardar é definida como TRUE, e o mutex é liberado;
    - Se o estado do buffer não estiver cheio, o produtor pode inserir o item no buffer.Sendo assim,o item é impresso na tela para indicar que está sendo inserido.Com isso,o estado do buffer é atualizado para BUFFERCHEIO e o mutex é liberado;

- O loop continua até que o produtor tenha produzido todos os 10 itens.

3. Funcão Consumidor:

   - Primeiramente um loop infinito é inicializado;
   - Dentro desse loop, um laço do-while é inicializado,e dentro dela, é chamada a função mutexLock, com o intuito de garantir que apenas um consumidor acesse o buffer por vez;
   - Em seguida, verifica-se se o estado do buffer é BUFFERVAZIO. Se o buffer estiver vazio, o consumidor precisa esperar, e a função mutexunlock(id) é chamada para liberar o mutex antes de esperar.Se o buffer não estiver vazio, significa que há um item disponível para consumir. Nesse caso, a variável item recebe o valor do item presente no buffer, e o estado do buffer é atualizado para BUFFERVAZIO, indicando que agora o buffer está vazio. Em seguida, a função mutexunlock(id) é chamada para liberar o mutex, permitindo que outros consumidores acessem o buffer;
   - Após retirar o item do buffer com sucesso, o consumidor processa o item;
   - O loop continuará indefinidamente, consumindo itens do buffer, até que uma chamada do Sistema Operacional seja chamada para interromper o programa.

# 4 Segundo Problema: Jantar dos Canibais

Para resolver o segundo problema, separamos a lógica em algumas classes do Java. As classes desenvolvidas foram: Canibal, Cozinheiro, LamportMutex, Semaphore e App (main). Abaixo se encontra a explicação de cada uma das classes utilizadas:

1. Canibal.java

   - Variáveis utilizadas:
     - id (para cada canibal);
     - porcoes (quantidade de porções);
     - mesa (semáforo que controla o acesso à mesa de comida);
     - mutex (implementação de mutex para garantir acesso exclusivo a mesa);
     - gerador de números aleatórios.
   - Método run
     - Um loop (while(true)) é executado continuamente enquanto o problema estiver em execução. Em seguida temos um try-catch para lidar com alguma exceção;
     - Dentro do try, fazemos o canibal esperar 3 segundos antes de começar a agir e, em seguida, é chamado o mutex.lock passando o id do canibal;
     - Dentro da condição if: se houver comida na mesa, é calculado quantas porções o canibal pode pegar (mínimo entre suas porções e a quantidade de comida na mesa) e é atualizado a quantidade de comida na mesa. Se não houver mais comida na mesa, o mutex é liberado (mutex.unlock(id)). Após o canibal pegar ou verificar a comida, o mesmo acontece (liberar o mutex) para permitir que outros canibais possam acessar a mesa;
     - O canibal se alimenta durante um período aleatório de tempo (entre 1 e 5 segundos).

2. Cozinheiro.java

- Variáveis utilizadas:
  - mesa (semáforo que controla o acesso à mesa);
  - porcoes (quantidade de porções que o cozinheiro prepara).
- Método run
  - Um loop (while(true)) é executado continuamente enquanto o problema estiver em execução. Em seguida temos um try-catch para lidar com alguma exceção;
  - O cozinheiro verfica continuamente (a cada 1 segundo) se ainda há comida na mesa;
  - Quando a mesa estiver vazia, o número de porções que o cozinheiro pode fazer é atualizado;
  - Após preparar a comida, o cozinheiro notifica os canibais (mesa.up()) que existem mais porções de comida.

3. LamportMutex.java

- Variáveis utilizadas
  - A classe implementa um algoritmo de exclusão mútua baseado no algoritmo de Lamport;
  - thread (array para armazenar o estado de cada thread/canibais);
  - thread = new int[num_threads] (cria um array com um tamanho igual ao numero de threads;
  - int maxVal (inicializa o valor maximo com o menor valor possivel).
- Método lock
  - Método que solicita acesso ao recurso (jantar).
- Método unlock
  - Método que libera acesso ao recurso.

4. Semaphore.java

- Variáveis utilizadas
  - value (estado atual do semáforo).
- Método up
  - O método é synchronized, garantido que apenas uma thread o execute por vez;
  - A variável value incrementa o valor do semáforo em 1;
  - Por fim, o notify notifica qualquer thread em espera que está aguardando neste semáforo.

5. App.java

- Variáveis utilizadas
  - canibais (informa o número de canibais para a execução do programa);
  - porcoes (informa o número de porções de comida para a execução do programa).
- Método Main
  - Inicializa o semáforo contador para a mesa;
  - Inicializa o mutex;
  - Cria as threads para os canibais e o cozinheiro;
  - O loop (for) inicializa cada uma das threads dos canibais;
  - Por fim, é criado um cozinheiro juntamente com a sua thread.