Sistemas Operacionais - Trabalho 1

Leonardo Vargas Schilling

Curso de Ciência da Computação
Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)

Porto Alegre – RS – Brazil

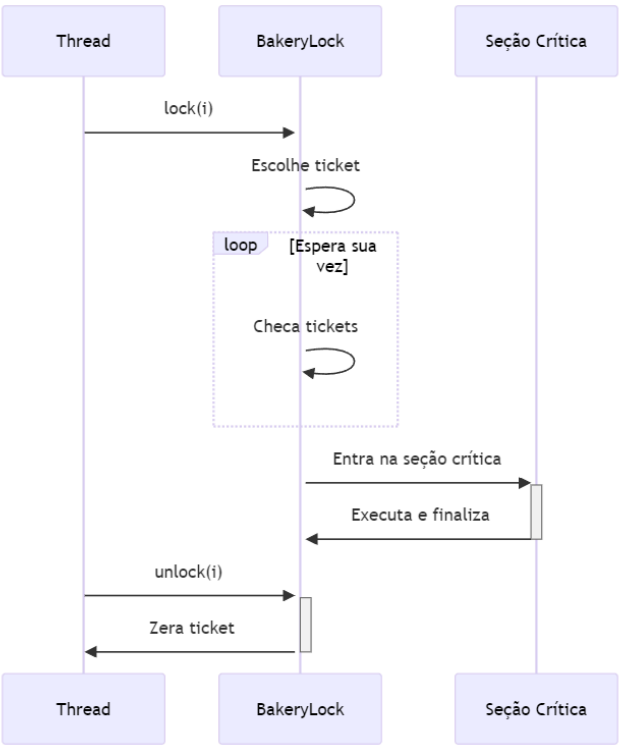
1. Introdução

Este relatório apresenta as soluções desenvolvidas para os problemas de Produtores e Consumidores e Jantar dos Canibais, conforme proposto no Trabalho 1. Ambos os problemas são abordados e solucionados por meio de um mecanismo que combina a sincronização baseada em eventos e o bloqueio de seções críticas utilizando uma implementação própria do algoritmo de padaria de Lamport.

2. Implementação do Algoritmo da Padaria de Lamport

Embora útil, é significativamente mais complexo generalizar o algoritmo de Peterson para N threads, portanto foi escolhido o algoritmo da padaria de Lamport, que, inspirado no sistema de senhas de uma padaria, assegura exclusão mútua permitindo que as threads acessem a seção crítica sequencialmente, baseando-se na ordem dos tickets que recebem.

Diagrama de sequência do algoritmo de Lamport



2.1 Detalhes de implementação do Algoritmo da Padaria

A implementação do algoritmo de Lampert está centralizada na classe ***BakeryLock***, que possui dois métodos, ***lock(thread_id)*** e ***unlock(thread_id)***. Além disso, possui os seguintes componentes:

total_threads: O número total de threads que precisam de sincronização.

choosing: Uma lista que controla threads no processo de obter um novo ticket.

ticket: Uma lista que armazena o ticket de cada thread.

O método ***lock(i)*** é invocado por uma thread para solicitar acesso à seção crítica, começa obtendo um *ticket*:

self.choosing[i] = True: Indica que a thread ***'i'*** está escolhendo seu ticket.

self.ticket[i] = max(self.ticket) + 1: Atribui à thread ***'i'*** o sucessor do maior ticket

self.choosing[i] = False: Indica que a thread ***'i'*** terminou de escolher seu ticket.

Após obter um ticket, a thread percorre a lista de todas as threads, comparando a thread ***'i'*** com todas as outras, denominadas por ***'o'***, e utiliza dois laços *while* que executam indefinidamente, até que o acesso seja liberado. Os laços de espera são avaliados da seguinte forma:

while self.choosing[o]: Verifica se outras threads estão escolhendo seus tickets.

(while self.ticket[o] != 0 and (self.ticket[o], o) < (self.ticket[i], i): Verifica a ordem dos tickets e usa o identificador da thread para resolver empates.

O método ***unlock(i)*** é chamado pela thread ***'i'*** para liberar a seção crítica, atribuindo zero ao seu ticket, permitindo assim que outras threads possam entrar na seção crítica.

3. Problema 1: Produtores e Consumidores

O problema foi modelado usando uma estrutura de dados de fila circular, representada pela classe ***CircularQueue***. Esta estrutura é implementada através de um vetor ***(self.queue)*** e referências para o início (***self.start***) e o fim ***(self.end)*** da fila, além de um contador ***(self.count)*** para manter o controle do número de itens na fila.

Para sincronizar as condições de fila cheia ou vazia e evitar espera ativa (***busy waiting***), utilizou-se a biblioteca ***threading*** do ***Python***, empregando ***threading.Event*** para os sinais ***not_full*** e ***not_empty*** ao consumir e produzir um ***item***, respectivamente. Isso garante que threads produtoras aguardem antes de tentar adicionar itens à fila cheia e que threads consumidoras façam o mesmo quando a fila está vazia. Os eventos são usados para bloquear as threads sob condições inadequadas, despertando-as somente quando a condição muda.

O método ***enqueue(item, thread_id)*** funciona da seguinte forma:

1. A thread produtora, identificada por ***thread_id***, tenta adquirir o lock.
2. Se a fila estiver cheia ***(count == size)***, a thread produtora libera o lock, limpa o evento ***not_full*** e entra em estado de espera até que o evento ***not_full*** seja sinalizado (indicando que a fila não está cheia).
3. Caso contrário, a thread adiciona o item à fila, atualiza o contador e os indicadores de posição e emite o evento ***not_empty*** para sinalizar que a fila não está vazia.

Já o método ***dequeue(item, thread_id)*** funciona da seguinte forma:

1. A thread consumidora, identificada por ***thread_id***, tenta adquirir o lock.
2. Se a fila estiver vazia ***(count == 0)***, a thread consumidora libera o lock, limpa o evento ***not_empty*** e entra em estado de espera até que o evento ***not_empty*** seja sinalizado (indicando que a fila não está vazia).
3. Caso contrário, a thread remove o item da fila, atualiza o contador e os indicadores de posição e emite o evento ***not_full*** para sinalizar que a fila não está cheia.

4. Problema 2: Jantar dos Canibais

O problema foi modelado considerando uma travessa compartilhada de comida como o recurso central, acessada por múltiplos canibais e um cozinheiro. A travessa é representada pela classe ***ServingPlatter***, que gerencia as porções de comida disponíveis e garante a sincronização entre os canibais (consumidores) e o cozinheiro (produtor).

Para alcançar essa sincronização, a biblioteca ***threading*** do ***Python*** foi utilizada, empregando os eventos ***not_empty*** para indicar que a travessa não está vazia e ***empty*** para sinalizar que a travessa está vazia e precisa ser abastecida.

Método ***eat(cannibal_id)*** na classe ***ServingPlatter***:

1. Aguarda até que haja porções disponíveis (***self.not_empty.wait***())
2. Adquire o lock (***self.mutex.lock(cannibal_id)***), garantindo acesso exclusivo
3. Consome uma porção, se disponível, decrementando o contador de porções. Se a travessa ficar vazia após a ação, sinaliza a condição ***(self.empty.set())*** para acordar o cozinheiro e limpa o evento ***self.not_empty*** para bloquear futuras tentativas de consumo até o reabastecimento.
4. Libera o lock (***self.mutex.unlock(cannibal_id)***), permitindo que outras threads acessem a travessa.

O método ***refill(),*** utilizado pelo cozinheiro, é responsável por abastecer a travessa:

1. O cozinheiro espera (***self.empty.wait()***) até a travessa estar vazia.
2. Adquire o lock (***self.mutex.lock(N)***), onde ***N*** é o identificador do cozinheiro.
3. Reabastece a travessa com o número máximo de porções (***M***) e sinaliza que há porções disponíveis (***self.not_empty.set()***). Simultaneamente, limpa o evento ***self.empty***.
4. Libera o lock ***(self.mutex.unlock(N))***, encerrando o abastecimento.