

Implementação Thread Safe dos Problemas Produtor/Consumidor e Jantar dos Canibais

André Sacilotto Santos¹, Florensa D'Ávila Dimer¹, Rodrigo de Oliveira Rosa¹

¹Escola Politécnica – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Caixa Postal 1429 – 90.619-900 – Porto Alegre – RS – Brazil

{andre.santos01,florensa.dimer,rodrigo.rosa81}@edu.pucrs.br

Abstract. *This report aims to provide an example of a thread-safe implementation of the producer-consumer problem and the dining cannibals problem in the C++ language, as well as to delve into its functionalities.*

Resumo. *Este relatório visa fornecer um exemplo de implementação thread safe do problema dos produtores e consumidores e jantar dos canibais na linguagem C++, assim como esmiuçar seu funcionamento.*

1. Visão Geral dos Problemas

O problema do produtor/consumidor apresenta um cenário onde é necessário implementar mecanismos para que os produtores e consumidores trabalhem com a mesma fila de comunicação. Nesse contexto, alguns problemas surgem ao considerar um fluxo de dados criado e consumido que pode exigir ordenação em algum momento durante a execução de aplicações reais.

Houve uma tentativa de alcançar essa situação de ordenação dos itens de dados na implementação da equipe. No entanto, tal proposta encontrou problemas relacionados a questões de granularidade fina ligadas à impressão das mensagens para o usuário na mesma ordem que foram produzidas no terminal.

O problema do jantar dos Canibais introduz requisitos adicionais de sincronização em comparação ao problema dos produtores/consumidores. Além de verificar se há comida para os canibais, a ocasião onde não há mais comida não resulta apenas em espera pelas *threads* canibais como no problema anterior.

Nesta situação, há um gatilho para que o canibal “acorde” a *thread* cozinheiro, que anteriormente encheu a travessa (fila de comunicação) e foi dormir. Este processo recorrente retoma o progresso dos fios de execução cada vez que o cozinheiro acorda a pedido de um canibal.

2. Implementação

A implementação de código foi feita na linguagem C++ usando o compilador g++. Neste contexto, a interface de programação paralela nativa C++ threads foi usada em conjunto com a biblioteca de semáforos herdada da linguagem C, dado o escopo e proposta do trabalho apresentados e discutidos em aula. A implementação de exclusão mútua se deu através do algoritmo de Peterson.

2.1. Mutex com algoritmo de Peterson

A implementação do algoritmo de Peterson foi feita para acoplar um *mutex* padrão (entre 2 threads) ou até mesmo com N threads. Para isso, dois vetores foram definidos no arquivo

“peterson.hpp”. Um armazena os níveis de profundidade das threads, análogos ao número esperado de threads (4 threads = 4 níveis), e o outro armazena o número de threads que está em cada nível.

Assim, quando uma thread tem sua entrada bloqueada na sessão crítica, verifica-se se nenhuma thread já está um nível acima ou igual, percorrendo o vetor de threads e seu nível, permitindo sua entrada na seção crítica e excluindo “mutuamente” todas as outras enquanto estiver nela.

2.2. Fila Bloqueante com semáforos

A implementação da fila circular bloqueante foi feita com dois semáforos. Um destes é chamado de “sem_empty” e é inicializado com a capacidade máxima padrão ou fornecida por parâmetro, enquanto outro chamado “sem_full” é inicializado com valores zerados indicando que a fila está vazia no estado inicial.

Por meio desses semáforos, é possível estabelecer quando o recurso da fila está vazio (disponível para inserção mas não para retirada), cheio (disponível para retirada porém não para inserção) ou não vazio nem cheio (disponível para ambas operações). Assim, a função enqueue() aguarda espaço na fila para inserção e sinaliza depois de inserir que há mais um elemento na fila, viabilizando a retirada do mesmo.

O método dequeue() performa operações parecidas porém de forma invertida sinalizando quando há espaço disponível na fila e aguardando até que haja elementos para retirada.

Por fim, funcionalidades de acesso à referências do vetor da fila circular também obedecem aos semáforos, aguardando até que haja elementos na fila para retornar a posição de front() e back().

2.3. Produtor e consumidor

A estratégia de implementação do produtor e consumidor se deu pela construção de um item de dados representado por uma struct Data que serviu para complementar a abstração dos consumidores.

Além disso, nas classes de cada estágio uma função sobrescrita operator() declara as instruções a serem executadas pelas threads.

Nessa função, no caso da classe Producer, um novo objeto com id inicial “0” é criado e anunciado através de mensagens ao usuário no terminal, sendo posteriormente inserido na fila de comunicação até que o último item de dados seja criado (especificado pelo usuário ou padrão de 1000 itens). Na classe Consumer por sua vez são retirados itens da fila e no caso de ser um dos últimos itens criado por um dos produtores, o consumidor finaliza.

Observa-se que se o usuário especificar um número de threads maior de consumidores do que produtores, os consumidores “extras” não finalizaram propriamente, necessitando que a aplicação seja terminada pelo usuário.

Para sincronizar as operações de escrita e leitura das informações foram utilizadas travas que permitiam a passagem de uma thread por vez na seção crítica do produtor e consumidor, assim como no acesso das informações da struct.

As informações da struct relativas a id, impressão, entre outras também necessitam de travas para que o acesso tanto de threads produtoras e consumidoras, que já estavam sendo executadas 1 por vez, não tivessem acesso concomitante aos dados da struct, gerando maior dificuldade de ordenação e terminação precisa das threads, baseadas no id do item de dados. Todas as travas utilizaram o algoritmo de Peterson.

2.4. Jantar dos Canibais

A implementação do problema do jantar dos canibais adotou um número arbitrário de threads canibais e porções de comida em sua entrada. Dessa forma, uma thread cozinheiro sempre foi instanciada, produzindo comida até encher a capacidade da travessa(fila de comunicação).

Ao atingir esse ponto, o cozinheiro vai dormir e desbloquear a trava dos canibais, os quais se deparam com outra trava que os permite comer um de cada vez através de outra trava definida com o número total de canibais passo por argumento pelo usuário e gerido com o id da thread canibal em questão.

Esse mecanismo possibilitou que o cozinheiro fosse acordado quando necessário quando um canibal destravar a trava dos mesmos(depois de dormir pela primeira vez) pela falta de comida. Este código roda indefinidamente até a interrupção do usuário. As travas usadas foram todas feitas com algoritmo de Peterson.

Essas travas poderiam ainda ser ajustadas em grão mais fino a fim de promover maior ordenação na saída do usuário de ambos papéis do problema, porém julgou-se fora dos escopo do trabalho, ficando como proposta futura de melhoria.

2.5. Joiner ThreadWrapper

Em ambas os problemas uma função *management* foi produzida a fim de empregar uma solução um pouco mais “elegante” para a terminação das threads. Basicamente é produzido um vetor do tipo da classe *threadWrapper*, que funciona como invólucro para thread a ser produzida.

Dentro desta implementação temos uma checagem da possibilidade de join já no momento em que esta é movida para dentro do vetor através de um operador de atribuição *move* que está declarado dentro da classe *threadWrapper*.

Além disso, quando o fluxo de execução da função *management* sai do laço que instancia as threads o destrutor do *wrapper* performa o join das threads depois estas voltam da função operador de cada uma das classes que lhe foram atribuídas.

3. Conclusão

Concluo que apesar de melhorias possíveis quanto a terminação pelo no problema dos produtores e consumidores, e na impressão para o usuário na saída de ambas implementações com foco para o jantar dos canibais a equipe alcançou um desempenho bastante satisfatório dado às propostas do enunciado deste trabalho, consolidando as abstrações teóricas e experimentais vistas em aula.