Relatório do segundo trabalho prático da disciplina Computação Concorrente

Matheus Fernandes[1], **Stephanie Orazem**[1]

[1] Departamento de Ciência da Computação
Universidade Federal do Rio de Janeiro (UFRJ) – Rio de Janeiro, RJ – Brazil

Link pro repositório no Github

matheusfcc2010@hotmail.com, sorazemhr@gmail.com

1. Introdução

Neste trabalho nos foi proposto realizar a implementação do problema concorrente Escritor-Leitor (*Reader-Writer*), em que as *threads* se dividem em **escritora** e **leitora**, sendo que as *threads* leitoras podem fazer leituras em conjunto de certo recurso quando nenhuma escritora estiver trabalhando, enquanto as *threads* escritoras podem escrever no recurso somente se estiver sozinha, ou seja, nenhuma outra *thread* do mesmo tipo pode estar trabalhando, e nenhuma *thread* leitora estiver lendo.

O recurso nesse trabalho é uma variável compartilhada entre todas as *threads*, na qual as escritoras escrevem seu número identificador e da qual as leitores lêem o valor e escrevem em seu respectivo arquivo de log.

Além disso, também deve ser gerado um arquivo de log da execução do programa e implementar um código que analise tal e verifique se o programa foi executado corretamente, seguindo os requisitos do problema.

2. Implementação

2.1. Ausência de Inanição do Leitor-Escritor

Para o desenvolvimento do trabalho, pensamos em uma fila de chegada para descobrir quais *threads* desejam acesso ao recurso e uma outra fila que garante o tal acesso exclusivo, que chamamos também de *Working Room*, onde ocorre a execução dos trabalhos das respectivas *threads*. **Importante ressaltar que a fila de chegada não é uma fila de ordem de chegada**.

Uma vez que uma *thread* tenta realizar sua função num momento em que nenhuma outra estiver trabalhando, ela garante acesso exclusivo da variável compartilhada para si, permite que outras *threads* passem da fila de chegada e a partir disso o fluxo de execução segue as condições do problema.

Se, por exemplo, uma leitora consegue tal acesso, ela permite que outras leitoras entrem e possam executar suas respectivas leituras da variável compartilhada até o momento em que uma escritora pede acesso. No momento em que ocorre isso, nenhuma leitora entra no *Working Room* e quando a última que entrou termina seu trabalho, ela libera o acesso exclusivo que é garantido logo em seguida pela escritora. E então, a escritora libera a fila de chegada novamente, que foi bloqueada anteriormente por ela, permitindo assim que uma *thread* passe e espere na fila do recurso, sendo a próxima a

trabalhar quando a escritora terminar de escrever. O fluxo de execução de quando uma *thread* escritora consegue o acesso primeiro funciona da forma que foi explicado anteriormente.

Utilizamos *mutex* para fazer as filas de chegada e do *Working Room* e, portanto, deixamos a mercê de quem conseguir acesso a CPU primeiro para executar. Se, por exemplo, uma escritora conseguiu acesso exclusivo ao recurso e nesse meio tempo entraram primeiro 3 leitoras e depois 1 escritora na fila de chegada, todas possuem a mesma chance de conseguir passar por essa fila quando a escritora liberar. Além dos *mutex* citados, utilizamos mais um para evitar condição de corrida na variável *reader*, que conta quantas leitoras estão lendo.

Entretanto, essa solução ainda é ineficiente, pois ainda há chance de, por exemplo, só leitoras ou só escritoras executarem, não porque o programa está dando prioridade a uma das duas mas sim porque ele dá abertura a essa possibilidade, por menor que seja. Além disso, percebemos nas execuções que ocorria muito de uma mesma *thread* executar várias vezes seguidas, o que significa que, por exemplo, uma leitora de número identificador 0 podia rodar muito mais vezes do que a de número identificador 1 numa situação de execução infinita. Apesar de na prática ambos os tipos executarem no mesmo ritmo, o código ainda era ineficiente devido a esses motivos apresentados.

Para solucionar tais problemas, adicionamos uma barreira para cada tipo de *thread* e, portanto, criamos 2 novos *mutex*, 2 variáveis de condição e 2 contadores de *threads* bloqueadas na barreira. Assim, uma *thread* depois de executar sua função só poderia o fazer novamente após todas as outras de seu mesmo tipo executarem também, o que não só impossibilita de uma mesma *thread* rodar várias seguidas como também garante que os dois tipos executem, pois quando última a *thread* de um tipo terminar de fazer seu trabalho, todas as outras certamente estarão bloqueadas na barreira e quando ela liberar o *Working Room* certamente uma *thread* do outro tipo irá garantir o acesso exclusivo à variável compartilhada.

Dessa forma, garantimos a ausência de inanição entre as *threads*.

2.2. Programa Auxiliar

Para a implementação do programa auxiliar utilizamos a linguagem **Python** para ter um desenvolvimento mais simples e por uma sugestão de solução da professora dada em sala de aula, apesar de não estarmos muito familiarizados.

Desenvolvemos a nossa solução no arquivo *auxiliar.py*. Durante a execução do leitor-escritor, ele copia o código do *auxiliar.py* para o arquivo passado por argumento (**que deve ter extensão '.py'**) e adiciona os logs no final, que estão em formatos de chamada de função para que chame as funções do *auxiliar.py* e assim ele possa verificar a corretude da execução. Fizemos dessa forma pois não encontramos nenhum outro jeito de poder gerar os logs em tais formatos e chamar as funções do programar auxiliar.

O conteúdo do arquivo é composto por dez funções que irão verificar a corretude da nossa solução, além de variáveis para manter atualizado o progresso da execução. Caso haja um erro, a mensagem referente a este é mostrada e o programa é terminado. Caso contrário, destacamos quando leitoras e escritoras realizam suas ações com sucesso e quando fazemos uso da fila de recurso.

Oferecemos mensagens de erro para as seguintes situações:

- Broadcast. Acontece quando uma thread leitora ou escritora mandou um sinal quando não deveria;
- Thread acessou o recurso quando não devia;
- Leitora leu o valor errado da variável compartilhada.

2.3. Problemas

No início da elaboração do código, partimos da ideia de criar uma fila de ordem de chegada, onde as *threads* fariam seu trabalho de acordo com a ordem que ela chegou nessa fila e respeitando os requisitos do problema. Entretanto, sentimos muita dificuldade em implementar essa solução, fazendo a gente gastar muito tempo e não tendo muito progresso. Então, nós pesquisamos algumas ideias pra dar alguma noção de por onde a gente poderia seguir e então achamos essa de duas filas, a qual desenvolvemos do nosso jeito utilizando *mutex* e barreiras.