Fundamentos de Sistemas Paralelos e Distribuídos
Primeiro exercício de programação: threads trabalhando em trios
Relatório

Diego Pereira da Silva
Matrícula: 2020006477

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

diegopsilva@ufmg.br

1. Apresentação

A atividade consiste em desenvolver um programa na linguagem C, usando Pthreads, para criar threads, cada uma com um determinado grupo especificado na entrada e então sincronizá-las em tarefas que devem ser realizadas em trio, cada trio contendo exatamente uma thread de cada grupo e com a restrição de haver apenas um trio em operação por vez. Além disso, as threads só poderão sair do trio quando todas as tarefas do trio estiverem terminadas, ou seja, uma thread não pode entrar em um trio que ainda está em execução, mesmo que a thread de seu grupo já tenha terminado sua tarefa no trio.

2. Organização da solução

A especificação já inclui os arquivos com as funções para simular as tarefas executadas pelas threads, que são simuladas chamando **spend_time**, essa função foi implementada nos arquivos spend_time.c e seu cabeçalho está disponível em spend_time.h. Além disso, a especificação prevê a criação das funções **init_trio**, **trio_enter** e **trio_leave**, que foram implementadas no arquivo trio.c com cabeçalho em trio.h. Por fim, a função principal que inclui a leitura dos dados e criação das threads foi implementada no arquivo main.c. As próximas seções detalham a implementação dos arquivos trio.c e main.c.

3. Trio

Esta é a parte principal da atividade, pois aqui especificou-se o comportamento ao entrar e sair do trio, que permite a sincronização seguindo as especificações do problema. Nas próximas subseções seguem detalhes da implementação da estrutura **trio_t** e de cada função desenvolvida.

3.1 Estrutura trio_t

É uma estrutura definida como “struct”em C para representar o trio, nela temos os seguintes atributos:

- Uma variável booleana **formado** para indicar que o trio já está formado, o que não necessariamente implica que a execução já começou, apenas que pode ter começado ou está apta a começar.
- Uma variável booleana **em_execucao** para indicar que o trio já está em processo de execução, ou seja, se há ao menos alguma thread executando uma tarefa nele.
- Um array de booleanos chamado **no_trio**, com três posições, cada uma representando se uma thread foi alocada para entrar no trio ou está no trio em execução. O índice **i** do array representa o grupo **i+1**. Se o valor atribuído é **1**, sabemos que uma thread do respectivo grupo já foi alocada para entrar no trio ou está no trio em execução. Já o valor **0** indica que o trio está em execução, mas a thread do grupo correspondente já terminou sua tarefa (nesse caso o valor e **em_execucao** será **1**), ou que um trio ainda não foi formado e ainda não há nenhum thread do respectivo grupo alocada para o trio que será formado.

Além disso, a estrutura armazenará os ponteiros para os objetos de pthreads necessários para realizar a sincronização, são eles:

- Um mutex **mutex_trio** para proteger a seção crítica (quando há acesso aos atributos do trio).
- Uma variável de condição **cond_formado** para sinalizar se o trio já foi formado.
- Uma variável de condição **cond_em_execucao** para sinalizar se o trio já acabou a execução.
- Um array de variáveis de condição **cond_no_trio**, cuja posição indica se uma thread está no trio, seguindo a mesma convenção adotada no array **no_trio**.

3.2 init_trio

Esta função recebe como parâmetro um ponteiro nomeado como **t**, que aponta para uma estrutura **trio_t**. Nesta função precisamos criar os objetos de pthreads, e como utilizamos ponteiros, aloca-se memória dinamicamente com a função **malloc**, e então chamamos **pthread_mutex_init** ou **pthread_cond_init** para inicializá-los. Por fim, atribuímos os valores iniciais para as variáveis **formado**, **em_execucao**, e cada posição de **no_trio**, inicialmente todos são falsos, e recebem, portanto, o valor **0**.

3.3 trio_enter

Esta função recebe como parâmetro um ponteiro nomeado como **t**, que aponta para uma estrutura **trio_t** e um inteiro **my_type**, que indica o grupo ao qual a thread que quer entrar no trio pertence. Inicialmente criamos uma variável **index** para armazenar o índice correspondente ao grupo, e assim acessarmos os arrays **no_trio** e **cond_no_trio** por esse índice. Em seguida, já travamos o **mutex_trio**, pois estaremos acessando as variáveis de **t**. Para que a thread possa entrar no trio, temos que garantir que o trio não esteja em execução e que não haja nenhuma thread alocada para entrar no trio, então usamos a função **pthread_cond_wait** para garantir que essas condições sejam satisfeitas, e então, entramos no trio, atribuindo o valor de **1** para a

posição de **no_trio** correspondente. No entanto, até o momento só garantimos que essa thread foi alocada para entrar no trio, mas o trio ainda não começou sua execução, pois para isso é necessário que as threads de todos os grupos tenham sido alocadas. Por isso, realizamos a verificação, e se todas as threads foram alocadas, atribuímos valor **1** à variável **formado** e sinalizamos através da respectiva variável de condição com a função **pthread_cond_broadcast**. Caso o trio não tenha sido formado, devemos esperar isso de fato acontecer para entrar na thread, para isso usamos novamente **pthread_cond_wait** com a variável **cond_formado**. Finalmente, o trio está em execução, então marcamos **em_execucao** como verdadeiro e podemos liberar a **mutex_trio** e terminar a função.

3.4 trio_leave

Esta função recebe como parâmetro um ponteiro nomeado como **t**, que aponta para uma estrutura **trio_t** e um inteiro **my_type**, que indica o grupo ao qual a thread que quer sair do trio pertence. Assim como em **trio_enter**, começamos criando a variável **index** e travando a **mutex_trio**. Então, já atribuímos o valor **0** à posição de **no_trio** correspondente e sinalizamos através de **cond_no_trio** usando a função **pthread_cond_signal**. No entanto, o trio pode ainda estar em execução, pois ele só termina, e a thread só pode deixar o trio, quando todas as threads acabaram suas tarefas, então checamos se este é o caso, e se for, marcamos que o trio já não mais está em execução, atribuindo **0** à variável **em_execucao**. E sinalizamos através da variável **cond_em_execucao** com **pthread_cond_broadcast**, nesse caso ela sinaliza que o trio encerrou sua execução. E caso o trio ainda não tenha terminado, deve-se esperar ele terminar para de fato deixá-lo, para isso chamamos **pthread_cond_wait** passando **cond_em_execucao** como parâmetro, e finalmente, o trio está dissolvido, não mais em execução e agora nem mesmo formado, portanto atribuímos **0** à variável **formado**, liberamos a **mutex_trio** e já encerramos a função.

3.5 destroy_trio

Esta função recebe como parâmetro um ponteiro nomeado como **t**, que aponta para uma estrutura **trio_t**. Sua implementação não foi explicitamente requisitada nas especificações, no entanto, como houve a alocação dinâmica de memória e criação de objetos de pthreads na função **init_trio**, é recomendado haver uma função para desfazer essas operações. Então esta função apenas chama **pthread_mutex_destroy** ou **pthread_cond_destroy** para destruir os objetos pthreads criados em **init_trio** e desaloca a memória alocada para os mesmos.

4. Main

No arquivo main.c importamos os cabeçalhos spend_time.h e trio.h, então criamos uma variável global do tipo **trio_t**, pois existe apenas um trio disponível segundo as especificações. Definimos uma estrutura com os parâmetros que passaremos à função executada pelas threads, que são os atributos recebidos na entrada: um id **tid**, o tipo (ou grupo) da thread **ttype**, o tempo que ela deve gastar executando sua tarefa sozinha **tsolo** e o tempo que ela deve gastar executando

sua tarefa em trio **ttrio**. Com isso criamos a função **executar_thread** com o funcionamento previsto na especificação, isto é, passa o tempo sozinha e depois entra no trio, passa o tempo no trio e sai do trio, para isso chamamos as funções **spend_time**, **trio_enter** e **trio_leave**. Já na função main propriamente dita, primeiro inicializamos o trio e guardamos as informações fornecidas na entrada, declaramos um array de **pthread_t** chamado **threads** e então chamamos **pthread_create** para as threads lidas, passando a função **executar_thread** e os argumentos adquiridos pela entrada padrão. Em seguida, chamamos **pthread_join** para esperar a finalização de cada thread e por fim, destruímos o trio chamando **destroy_trio**.

5. Conclusão

Ao final, o programa funcionou como esperado, ao menos nos casos testados, e foi possível realizar a sincronização utilizando mutex e variáveis de condição, como recomendado na especificação, sem o uso de semáforos, que requereria uma solução mais complexa. Desse modo, foi possível colocar em prática os conceitos adquiridos na disciplina de Fundamentos de Sistemas Paralelos e Distribuídos, consolidando o aprendizado.

Bibliografia

GUEDES, Dorgival O. (2023). Slides virtuais da disciplina de Fundamentos de Sistemas Paralelos e Distribuídos.

Tanenbaum, A.S. and Steen, M.van (2007) *Distributed systems: Principles and paradigms*. Upper Saddle River, NJ: Prentice Hall.

C Reference. Disponível em: <https://en.cppreference.com/w/c>. Acesso em: 30 de abril de 2023.