Universidade Federal do Rio Grande do Sul - Instituto de Informática
Departamento de informática aplicada
INF01151 – Sistemas Operacionais II N

ATIVIDADE DE PROGRAMAÇÃO:

EXCLUSÃO MÚTUA IMPLEMENTADA EM SOFTWARE

Kauê dos Santos Gomes (00302416)

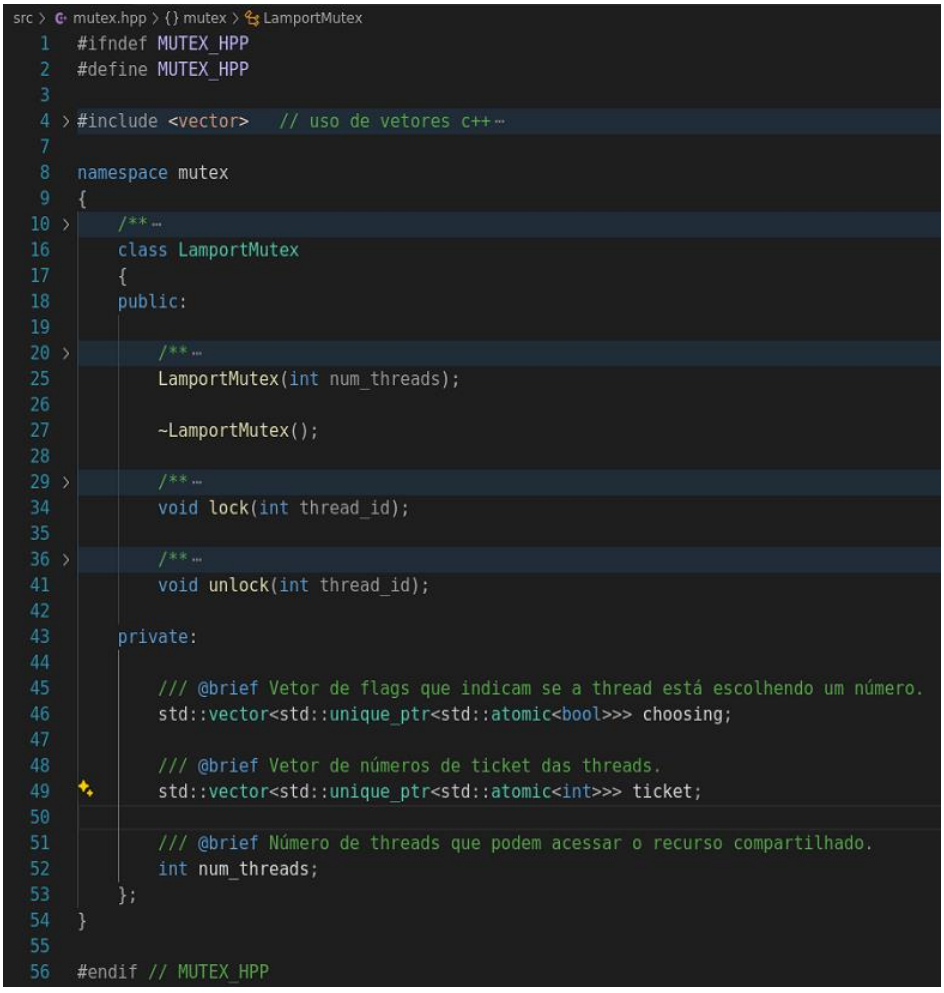
Porto Alegre, 14 de abril de 2024

Esta atividade de Programação tem por objetivo realizar a implementação de primitivas de exclusão mútua em software, usando como estudo de caso o algoritmo de Lamport para resolver o problema de exclusão mútua (também conhecido como “Algoritmo da Padaria”).

A atividade foi desenvolvida usando a linguagem C++, em ambiente Linux. Para aproveitar os recursos da linguagem, a implementação foi realizada utilizando namespaces e classes, implementando os métodos lock() e unlock() para garantir que o acesso à seção crítica do código seja realizado uma única vez por thread utilizando o algoritmo de Lamport.

DESENVOLVIMENTO

Para criação da biblioteca compartilhada sugerida nesta atividade, foi utilizado a linguagem C++ e seus recursos, implementando a classe ***mutex::LamportMutex*** onde há os métodos ***lock()*** e ***unlock()***, fazendo utilização do algoritmo de Lamport.



Foi utilizado a classe ***std::atomic***, da biblioteca atomic de C++, para garantir que os processos de leitura e escrita nas variáveis do vetor ***choosing*** e ***ticket*** sejam realizada

de forma atômica, ou seja, sejam executadas de uma única vez pelo processador, com garantia de que não haverá interrupções durante esses processos.

A utilização da classe ***std::unique_ptr***, da biblioteca memory de C++, foi necessária pois ***std::atomic*** não permite cópias da classe, excluído na implementação do objeto, e ***std::vector*** precisa que seus elementos possam ser copiados para o seu funcionamento na forma como foi implementado. Desta forma, quando ***std::vector*** realiza cópias dos seus elementos ele está realizando cópias de ponteiros para objetos.

O método sugerido para inicialização, ***lamport_mutex_init()***, foi substituído pelo constructor da classe ***mutex::LamportMutex***, uma vez que esta recebe o número de threads que o algoritmo suportará e inicializa os vetores ***choosing*** e ***ticket***.

src > mutex.cpp > ...

```cpp
#include "mutex.hpp"

mutex::LamportMutex::LamportMutex(int num_threads)
{
    this->num_threads = num_threads;

    // Inicializa os vetores de flags e de números de ticket das threads
    for (int i = 0; i < num_threads; i++)
    {
        this->choosing.push_back(std::make_unique<std::atomic<bool>>(false));
        this->ticket.push_back(std::make_unique<std::atomic<int>>(0));
    }
}
```

Os métodos ***lamport_mutex_lock()*** e ***lamport_mutex_unlock()*** foram substituídos pelos métodos membro ***lock()*** e ***unlock()***, da classe ***mutex::LamportMutex***.

No método ***lock()*** foi implementado o algoritmo de Lamport onde a thread é capaz de informar que irá escolher uma ficha, escolher uma ficha e aguardar a liberação para a seção crítica (lógica para garantia de exclusão mútua do algoritmo de Lamport).

```cpp
void mutex::LamportMutex::lock(int thread_id)
{
    // Marca a thread atual como escolhendo um número
    this->choosing[thread_id]->store(true);

    // Escolhe o maior número de ticket dentre todas as threads
    int max_ticket = 0;
    for (int i = 0; i < this->num_threads; i++)
    {
        int current_ticket = this->ticket[i]->load();
        max_ticket = (current_ticket > max_ticket) ? current_ticket : max_ticket;
    }

    // Atribui um número de ticket maior que o maior número de ticket dentre todas as threads
    this->ticket[thread_id]->store(max_ticket + 1);

    // Marca a thread atual como não escolhendo um número
    this->choosing[thread_id]->store(false);

    // Aguarda até que todas as threads tenham escolhido um número
    for (int i = 0; i < this->num_threads; i++)
    {
        while (this->choosing[i]->load()) {}
        while ((this->ticket[i]->load() != 0) &&
               ((this->ticket[i]->load() < this->ticket[thread_id]->load()) ||
               ((this->ticket[i]->load() == this->ticket[thread_id]->load()) &&
               (i < thread_id)))) {}
    }
}
```

Enquanto no método ***unlock()*** é realizado a liberação da seção crítica.

```
47  void mutex::LamportMutex::unlock(int thread_id)
48  {
49      // Marca o número de ticket da thread atual como 0
50      this->ticket[thread_id]->store(0);
51  }
```

COMPARATIVOS

Para validação da biblioteca desenvolvida e comparação com a implementação de exclusão mútua da biblioteca ***pthread***, foram desenvolvidos os softwares ***thread_race_cond_lamport*** e ***thread_race_cond_pthread***.

Ambos softwares foram desenvolvidos para que 3 threads incrementem 3.000.000 vezes uma variável global que inicialmente tem seu valor atribuído como 0, dessa forma, se o princípio de exclusão mútua funcionar, ambos softwares devem terminar sua execução com a variável global definida com o valor 9.000.000.

Foi utilizado a função built-in ***time***, do *bash*, para realizar os comparativos entre os tempos de execução das diferentes implementações.

Resultados da execução do software ***thread_race_cond_lamport***:

```
kaue@kaue-Vostro-3583:~/Projetos/PG1_20241/example$ g++ -L . thread_race_cond_lamport.cpp -o thread_race_cond_lamport  -lmutex
kaue@kaue-Vostro-3583:~/Projetos/PG1_20241/example$ time LD_LIBRARY_PATH=. ./thread_race_cond_lamport
9000000

real    0m2,562s
user    0m7,622s
sys     0m0,007s
kaue@kaue-Vostro-3583:~/Projetos/PG1_20241/example$
```

Resultados da execução do software ***thread_race_cond_pthread***:

```
kaue@kaue-Vostro-3583:~/Projetos/PG1_20241/example$ g++  thread_race_cond_pthread.cpp -o thread_race_cond_pthread  -lpthread
kaue@kaue-Vostro-3583:~/Projetos/PG1_20241/example$ time ./thread_race_cond_pthread
9000000

real    0m0,513s
user    0m0,732s
sys     0m0,623s
kaue@kaue-Vostro-3583:~/Projetos/PG1_20241/example$
```

Observando os resultados acima, é possível verificar que o algoritmo de Lamport implementado nesta atividade é consideravelmente mais lento que a implementação do mutex de *phtread*, isso ocorre pois:

- O algoritmo de Lamport é implementado no nível do usuário, enquanto mutex de *pthread* é implementado no nível do sistema operacional. Isso significa que o algoritmo desenvolvido precisa de mais código e lógica para garantir a exclusão mútua, o que pode levar a um overhead de implementação.
- O algoritmo de Lamport usa uma técnica chamada *busy waiting*, onde um thread em espera consome CPU até que obtenha acesso ao recurso. Isso pode ser

ineficiente em termos de uso de CPU. Por outro lado, *pthread_mutex* coloca threads em espera para dormir, economizando CPU.

- O Algoritmo da Padaria pode não escalar bem para um grande número de threads devido ao seu design. Cada thread precisa verificar o status de todas as outras threads, o que pode ser caro em termos de tempo para um grande número de threads. Por outro lado, mutex de *pthread* pode lidar melhor com escalabilidade por utilizar recursos do sistema operacional para garantir a exclusão mútua.

Entretanto, devido ao fato da implementação do algoritmo de Lamport utilizar espera ativa para garantir a exclusão mútua, sua utilização de recursos do sistema (valor descrito como sys no resultado da função time do bash) é menor quando comparado com a implementação de mutex de *pthread*.

DIFICULDADES

Implementar o algoritmo de padaria de Lamport é desafiador por si só, entretanto, com o pseudocódigo fornecido nas aulas a complexidade foi reduzida.

A escolha da utilização de abordagens mais modernas, como a utilização de ***std::vector***, ***std::atomic*** e ***std::unique_ptr***, de C++ tornaram o desenvolvimento do algoritmo um pouco mais complexa e provavelmente mais lenta quando comparado com uma implementação utilizando de recursos primitivos, como vetores clássicos de C, por exemplo.

COMPILAÇÃO

Para facilitar na compilação da biblioteca e os arquivos de testes desse projeto, foi implementado um Makefile, sendo necessário apenas executar os comandos ***make*** para compilar a biblioteca e os testes e executar ***make clean*** caso deseja limpar os arquivos compilados.

OBS.: Os comandos citados anteriormente e posteriormente foram projetados considerando que o local de execução é a raiz do projeto em ambiente Linux.

Após executar o comando ***make***, você pode executar os softwares de teste usando os respectivos comandos:

- ***LD_LIBRARY_PATH=./lib ./build/thread_race_cond_lamport***
- ***./build/thread_race_cond_pthread***

O resultado esperado dos dois arquivos de teste é o valor 9.000.000 (resultado de 3 threads executando uma função que incrementa 3.000.000 vezes uma unidade em uma variável cuja qual tem seu valor inicial como zero).

Para executar os testes com mensuração dos tempos de execução de cada um, deve-se executar os seguintes comandos:

- ***time LD_LIBRARY_PATH=./lib ./build/thread_race_cond_lamport***
- ***time ./build/thread_race_cond_pthread***

O resultado esperado das duas execuções acima é o resultado dos softwares de teste (9.000.000) e a mensuração real, de usuário e de sistema da função build-in time do bash do linux.

ESTRUTURA DE ARQUIVOS

Após a execução do comando ***make***, o projeto apresenta a seguinte estrutura de arquivos:

```
.
├── build
│   ├── thread_race_cond_lamport
│   └── thread_race_cond_pthread
├── example
│   ├── thread_race_cond_lamport.cpp
│   └── thread_race_cond_pthread.cpp
├── lib
│   └── libmutex.so
├── Makefile
├── relatorio.pdf
└── src
    ├── mutex.cpp
    └── mutex.hpp
```