Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет информационных технологий и управления

Дисциплина: “Операционные системы”

# Лабораторная работа №3 по теме:
# “Алгоритмы синхронизации процессов”

Выполнили:
Кимстач Д.Б.
Заломов Р. А.
Проверил:
Цирук В. А.

Минск 2022

Цель: Изучить средства синхронизации процессов/потоков и средства операционной системы, позволяющей осуществлять синхронизацию.

Задача:

*Обедающие философы (вариация). Создается пять процессов (по одному на философа). Процессы разделяют пять переменных (вилок). Каждый процесс находится только в двух состояниях - либо он "размышляет", либо "ест спагетти". Чтобы начать "есть", процесс должен взять "две вилки" (захватить две переменные). Закончив "еду", процесс освобождает захваченные переменные и начинает "размышлять" до тех пор, пока снова "проголодается".*

Решение:

В качестве способа синхронизации был выбран “mutex”.

**Мью́текс** (англ. *mutex*, от *mutual exclusion* — «взаимное исключение») — примитив синхронизации, обеспечивающий взаимное исключение исполнения критических участков кода.

Как и сказано выше, мьютекс исключает возможность использование критических участков кола несколькими процессами/потоками. Исходя из условия задачи (смотри выше), можно заметить, что одна вилка (переменная) не может быть использована одновременно несколькими философами (потоки, процессы). Поэтому для способа синхронизации был задействован упомянутый подход (мьютекс).

Программный код:

Файл main.cpp:

```cpp
#include <iostream>
#include "source/dining_philosophers.h"

int main(int argc, char** argv)
{
    std::cout << "Adding philosophers...\n";
    //adding 5 philosophers
    add_phil( name: "Plato",  think_t: 1,  eat_t: 3);
    add_phil( name: "Kant",  think_t: 2,  eat_t: 2);
    add_phil( name: "Nietzsche",  think_t: 3,  eat_t: 1);
    add_phil( name: "p1",  think_t: 1,  eat_t: 3);
    add_phil( name: "p2",  think_t: 2,  eat_t: 3);
    std::cout << "Philosophers added. Starting dinner...\n";

    start_dinner( time_ms: 30);

    return 0;
}
```

Данный файл лишь импортирует все реализованные классы из папки “source”. Так же в нем идет инициализация все философов, их время для размышления, еды и общее время ужина.

Файлы stick.h/stick.cpp:

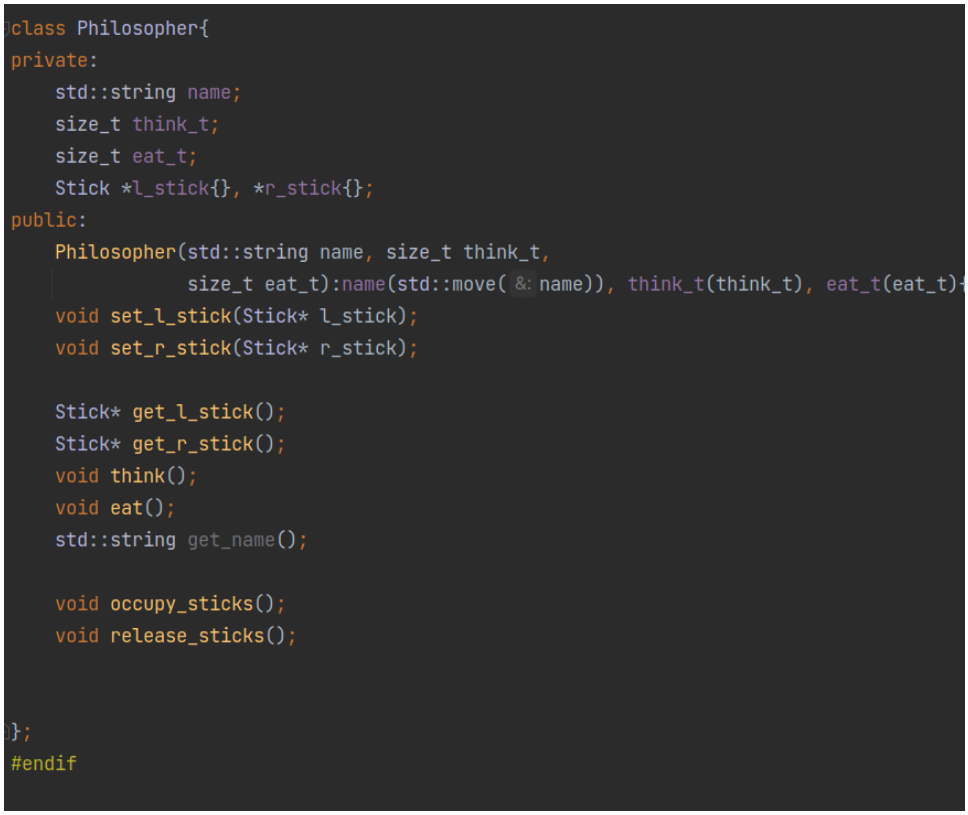
```
//
// Created by daniil on 19.10.22.
//

#ifndef OS_LAB_3_STICK_H
#define OS_LAB_3_STICK_H
#include <mutex>

class Stick{
public:
    void unlock();
    bool try_lock();
private:

    //we have only the object of the class 'mutex'
    std::mutex m;
};

#endif //OS_LAB_3_STICK_H
```

```
#include "stick.h"
//There only two methods:
//1. unlock - unlocking the memory area by the thread
//2. try_lock - trying to occupy the memory area
void Stick::unlock() {
    m.unlock();
}

bool Stick::try_lock() {
    return m.try_lock();
}
```

В данных файлах осуществляется объявление и реализация класса “Stick”. Stick - переменная, критическая область памяти. Так же данный класс имеет объект “mutex”, два метода: “unlock”, “try_lock”. Первый метод необходим для освобождения области памяти потоком/процессов. Второй

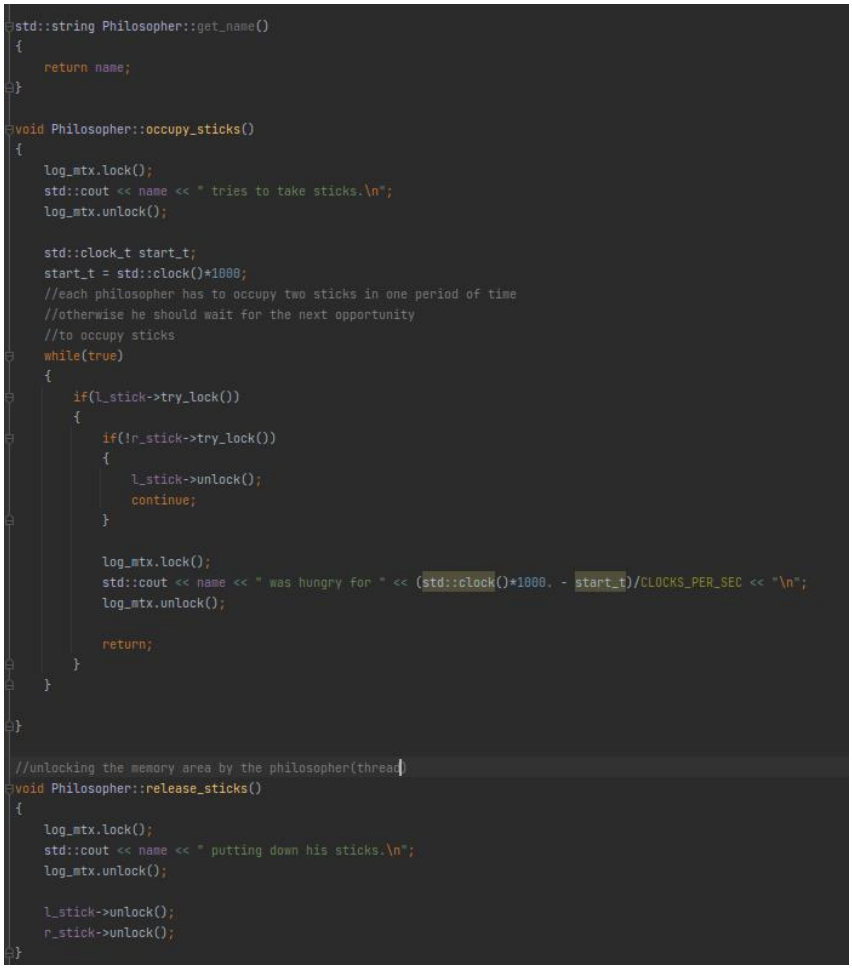
возвращает логическое значение о успехе/провале “захвата” области памяти.

Файлы philosophers.h/philosophers.cpp:



```
class Philosopher{
private:
    std::string name;
    size_t think_t;
    size_t eat_t;
    Stick *l_stick{}, *r_stick{};
public:
    Philosopher(std::string name, size_t think_t,
                size_t eat_t):name(std::move(&name)), think_t(think_t), eat_t(eat_t){
    void set_l_stick(Stick* l_stick);
    void set_r_stick(Stick* r_stick);

    Stick* get_l_stick();
    Stick* get_r_stick();
    void think();
    void eat();
    std::string get_name();

    void occupy_sticks();
    void release_sticks();


};
#endif
```

```cpp
#include "philosophers.h"
#include <iostream>
#include <ctime>
#include <mutex>
#include <unistd.h>

static std::mutex log_mtx;

void Philosopher::set_l_stick(Stick* l_stick)
{
    this->l_stick = l_stick;
}

void Philosopher::set_r_stick(Stick* r_stick)
{
    this->r_stick = r_stick;
}

Stick* Philosopher::get_l_stick()
{
    return l_stick;
}

Stick* Philosopher::get_r_stick()
{
    return r_stick;
}

void Philosopher::think()
{
    //we have to lock the definite memory are to write something into
    //the console
    log_mtx.lock();
    std::cout << name << " is thinking now\n";
    log_mtx.unlock();

    usleep( useconds: think_t*1000);
}

void Philosopher::eat()
{
    log_mtx.lock();
    std::cout << name << " is eating now\n";
    log_mtx.unlock();

    usleep( useconds: eat_t*1000);
}
```



```cpp
std::string Philosopher::get_name()
{
    return name;
}

void Philosopher::occupy_sticks()
{
    log_mtx.lock();
    std::cout << name << " tries to take sticks.\n";
    log_mtx.unlock();

    std::clock_t start_t;
    start_t = std::clock()*1000;
    //each philosopher has to occupy two sticks in one period of time
    //otherwise he should wait for the next opportunity
    //to occupy sticks
    while(true)
    {
        if(l_stick->try_lock())
        {
            if(!r_stick->try_lock())
            {
                l_stick->unlock();
                continue;
            }

            log_mtx.lock();
            std::cout << name << " was hungry for " << (std::clock()*1000. - start_t)/CLOCKS_PER_SEC << "\n";
            log_mtx.unlock();

            return;
        }
    }

}

//unlocking the memory area by the philosopher(thread)
void Philosopher::release_sticks()
{
    log_mtx.lock();
    std::cout << name << " putting down his sticks.\n";
    log_mtx.unlock();

    l_stick->unlock();
    r_stick->unlock();
}
```

В данном классе реализуется класс “Философ”. Каждый философ имеет имя, время для размышления, время для еды, правую и левую вилку (переменную). Также здесь реализован цикл “while” для “захвата” переменных. Хотелось бы заметить, что философ может захватить только 2 вилки для начала трапезы, но не одну.

Файлы dining_philisophers.h/dining_philosophers.cpp:

Главный функционал программы реализован именно здесь.

```
//

#ifndef OS_LAB_3_DINING_PHILOSOPHERS_H
#define OS_LAB_3_DINING_PHILOSOPHERS_H

#include <vector>
#include <string>
#include <stdlib.h>
#include "philosophers.h"

static std::vector<Philosopher> phils;

// add_phil adds a philosopher in vector 'phils'.
void add_phil(const std::string& name, int think_t, int eat_t);

// start_dinner starts threads of dining philosophers.
void start_dinner(size_t time_ms);
#endif //OS_LAB_3_DINING_PHILOSOPHERS_H
```

```
#include <iostream>
#include <unistd.h>

void add_phil(const std::string& name, int think_t, int eat_t)
{
    //emplace_back is something like constructor.
    //In this method we create an attribute of the class
    //Philosopher with all the needed values
    std::cout << "Adding " << name << "\n";;
    phils.emplace_back(name, think_t, eat_t);
    //creating new sticks for the first philosopher
    if(phils.size() == 1)
    {
        phils[0].set_l_stick(new Stick);
        phils[0].set_r_stick(new Stick);

        return;
    }
    //we should have only 5 sticks
    if(phils.size() == 5)
    {
        phils[4].set_l_stick((phils[3]).get_r_stick());
        phils[4].set_r_stick(phils[0].get_l_stick());
        return;
    }

    std::vector<Philosopher>::iterator phil = phils.end();

    (phil - 1)->set_l_stick((phil - 2)->get_r_stick());
    (phil - 1)->set_r_stick(new Stick);

}

void start_phil(size_t i)
{
    while(true)
    {
        phils[i].think();
        phils[i].occupy_sticks();
        phils[i].eat();
        phils[i].release_sticks();
    }
}
```

```cpp
void start_dinner(size_t time_ms)
{
    std::vector<std::thread> p_threads;
    for(size_t i = 0; i < phils.size(); i++)
    {
        p_threads.push_back(std::thread( & start_phil, i));
    }

    std::thread timer( & usleep, time_ms*1000);
    //join() waits until a thread completes
    timer.join();
    //detach() is mainly useful when you have a task that has to be done in background, but you don't care about it
    // This is usually a case for some libraries. They may silently create a background worker
    // thread and detach it so you won't even notice it.
    for(std::thread & th : p_threads)
    {
        th.detach();
    }

    std::cout << "Dinner finished";
}
```

В данных файлах идет выделение пяти для 5 вилок (переменных). Так же именно здесь начинается “ужин”.

Результат выполнения программы:

```
Adding philosophers...
Adding Plato
Adding Kant
Adding Nietzsche
Adding p1
Adding p2
Philosophers added. Starting dinner...
Plato is thinking now
Kant is thinking now
Nietzsche is thinking now
p1 is thinking now
p2 is thinking now
Plato tries to take sticks.
Plato was hungry for 0.002
Plato is eating now
p1 tries to take sticks.
p1 was hungry for 0.001
p1 is eating now
Kant tries to take sticks.
p2 tries to take sticks.
Nietzsche tries to take sticks.
Plato putting down his sticks.
Plato is thinking now
p1 putting down his sticks.
p1 is thinking now
p2 was hungry for 5.208
p2 is eating now
Kant was hungry for 5.258
Kant is eating now
Plato tries to take sticks.
p1 tries to take sticks.
Kant putting down his sticks.
Nietzsche was hungry for 8.879
Nietzsche is eating now
Kant is thinking now
p2 putting down his sticks.
p2 is thinking now
Plato was hungry for 4.867
Plato is eating now
Nietzsche putting down his sticks.
Nietzsche is thinking now
p1 was hungry for 5.023
p1 is eating now
Kant tries to take sticks.
p2 tries to take sticks.
Dinner finished
Process finished with exit code 0
```

Как можно видеть, ни один философ не подвергся длительному “голоданию”. Программа была завершена успешно (Proccess finished with exit code 0).

Вывод: в ходе выполнения лабораторной работы мы ознакомились с алгоритмом решения классической зачади на синхронизацию процессов/потоков, также мы познакомились с инструментами синхронизации Операционной системы (mutex).