Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

**ОТЧЁТ**

к лабораторной работе

на тему

Взаимодействие процессов (потоков): взаимное исключение и синхронизация

Выполнил: студент группы 153505

Кудласевич Артур Иванович

Проверил: Сиротко Сергей Иванович

Минск 2023

СОДЕРЖАНИЕ

[1 Постановка задачи 3](#_Toc146631498)

[2 Краткие теоретические сведения 4](#_Toc146631499)

[3 Результаты выполнения лабораторной работы 6](#_Toc146631500)

[Выводы 7](#_Toc146631501)

[Приложение А (обязательное) Листинг кода 8](#_Toc146631503)

**1 ПОСТАНОВКА ЗАДАЧИ**

Целью лабораторной работы является изучение подходов, системных объектов и функций, для обеспечения синхронизации и передачи управления между взаимодействующими процессами. А также изучение типичных проблем, возникающих при организации взаимодействия, моделей для их описания, путей их решения.

**2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Предположим, что в ВС более двух процессов, выполняющих некоторую последовательность программных инструкций в бесконечном цикле. Выполняемые инструкции разделяются на четыре непрерывных блока кода: код, не требующий синхронизации, КС, вход в КС, выход из КС. Каждый процесс начинает свое выполнение с блока кода, не требующего синхронизации. В некоторый момент времени процессу может потребоваться выполнить инструкции из блока критической секции. Перед этим ему необходимо войти в КС, выполнив инструкции блока входа в КС, тем самым обеспечив себе эксклюзивное право выполнения инструкций КС. По завершении КС процесс выполняет инструкции выхода из нее, тем самым оповещая другие процессы о возможности получения доступа к КС. После выхода из КС процесс продолжает выполнять код, не требующий синхронизации.

Задача взаимного исключения состоит в разработке кода для блоков входа в КС и выхода из КС таким образом, чтобы были удовлетворены два основных требования:

*1. взаимное исключение* (англ. mutual exclusion) — свойство безопасности, которое гарантирует, что никакие два процесса не могут одновременно находиться внутри КС;

2. *отсутствие мертвой блокировки* или *свобода от взаимоблокировок* (англ. deadlock freedom) — свойство живости, подразумевающее, что если один процесс пытается войти в КС, то некоторый процесс, не обязательно тот же самый, рано или поздно войдет в свою КС.

Условие отсутствия мертвой блокировки предоставляет *гарантию прогресса* (живость) всей системы в целом, однако оно допускает ситуацию “голодания” отдельных процессов, которые периодически и безуспешно пытаются войти в КС. Более строгое условие гарантии прогресса — условие отсутствия “голодания” процессов (англ. starvation freedom) — если один процесс пытается войти в КС, то этот процесс рано или поздно обязательно войдет в свою КС. Для перечисленных условий решение задачи взаимного исключения может гарантировать определенную справедливость порядка выполнения КС процессами. Примером справедливого условия может служить условие “первый зашел-первым обслужен” или FIFO-справедливость (англ. first-in-first-out) — ни один процесс не может войти в КС до процесса, уже ожидающего своей очереди на вход в КС.

Изначально проблема взаимного исключения ставилась для систем с общей памятью, в которых есть возможность применения атомарных на аппаратном уровне операций чтения/записи. В настоящее время существует большое количество методов и средств решения данной задачи как для мультипроцессорных, так и для мультикомпьютерных систем.

Методы бесконфликтного доступа к памяти, которые решают задачу взаимного исключения, основаны на понятии блокировки или исключающего права. Попытка захвата блокировки осуществляется при входе в КС, а освобождение — при выходе из КС. При безуспешной попытке захвата процесс останавливается и помещается операционной системой в очередь ожидания блокировки.

Обладание таким исключающим правом означает для процесса возможность исполнения КС. Популярность данного подхода обосновывается относительной простотой использования средств синхронизации на основе блокировок при программировании и наличием достаточно эффективных реализаций блокировок.

**3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ     РАБОТЫ**

В результате выполнения лабораторной работы было создано приложение, которое реализует модель взаимодействия конкурирующих параллельных процессов (потоков) “обедающие философы”.

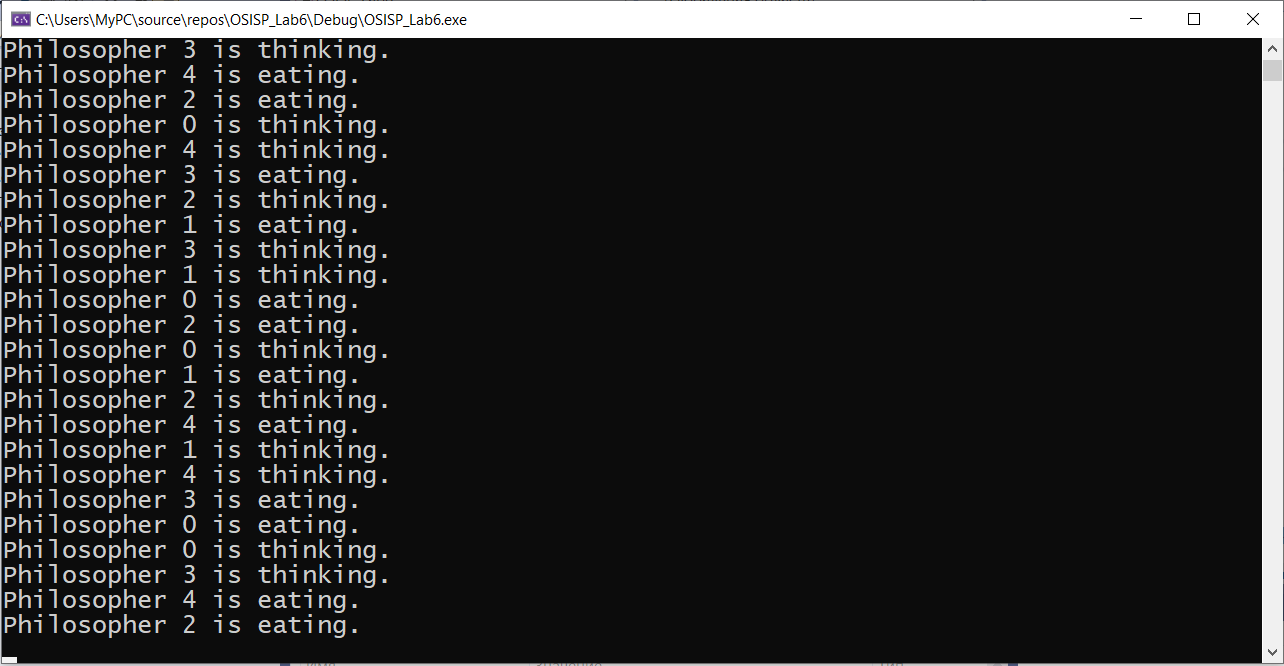


Рисунок 1 – Результат работы программы

**ВЫВОДЫ**

В результате выполнения данной лабораторной работы была разработана программа с использованием C++, которая реализует модель взаимодействия конкурирующих параллельных процессов (потоков) “обедающие философы” с наглядным представлением результатов.

**ПРИЛОЖЕНИЕ А**

## **(обязательное)**

**Листинг кода**

#include <iostream>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <vector>

const int NUM\_PHILOSOPHERS = 5;

std::mutex forks[NUM\_PHILOSOPHERS];

std::condition\_variable cond\_vars[NUM\_PHILOSOPHERS];

void philosopher(int id) {

int left\_fork = id;

int right\_fork = (id + 1) % NUM\_PHILOSOPHERS;

while (true) {

std::unique\_lock<std::mutex> left\_lock(forks[left\_fork]);

std::unique\_lock<std::mutex> right\_lock(forks[right\_fork]);

std::cout << "Philosopher " << id << " is eating." << std::endl;

// Simulate eating

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));

right\_lock.unlock();

left\_lock.unlock();

std::cout << "Philosopher " << id << " is thinking." << std::endl;

// Simulate thinking

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));

}

}

int main() {

std::vector<std::thread> philosophers;

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

philosophers.emplace\_back(philosopher, i);

}

for (auto& philosopher : philosophers) {

philosopher.join();

}

return 0;

}