Министерство Образования, Культуры,

Исследований Республики Молдова

Технический Университет Молдовы

Департамент Программная Инженерия и Автоматика

**Отчёт**

по лабораторной работе №1

**по дисциплине «PR»**

Выполнил: ст.гр. TI-197

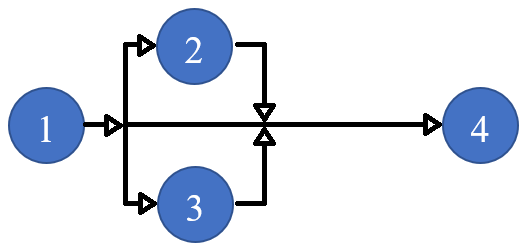
Шарафудинов Н.

Проверил: Anghiloglu A.

Кишинёв - 2022

**Тема:** Многопоточность

**Цель работы:** понимание моделей параллельного исполнения и знание основных методов синхронизации действий на основе атомарных операций светофора.



**1) Что такое многопоточное программирование ?**

Многопоточность — свойство платформы (например, операционная система, виртуальная машина и т. д.) или прикладное программное обеспечение/приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющих параллельные вычисления, то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

**2) Какие виды многопоточности бывают.**

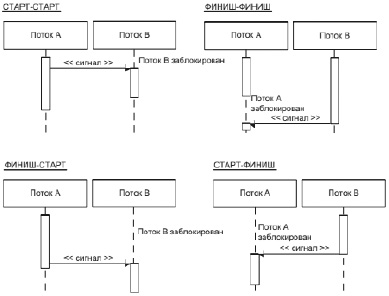
Различают две формы многопоточности, которые могут быть реализованы в процессорах аппаратно:

• Временная многопоточность (англ. Temporal multithreading)

• Одновременная многопоточность (англ. Simultaneous multithreading)

**3) Взаимодействие потоков.**

Выделяют 4 общих типа (рисунок 1) синхронизации любых двух потоков в одном процессе или любых двух процессов в одном приложении: старт-старт, финиш-старт, старт-финиш и финиш-финиш. С помощью этих базовых типов отношений можно описать координацию задач между потоками и процессами.



Рисунок

Возможные варианты синхронизации потоков/процессов

* Синхронизация типа старт-старт. Одна задача может начаться раньше другой, но не позже.
* Синхронизация типа финиш-старт задача A не может завершиться до тех пор, пока не начнется задача B. Этот тип отношений типичен для процессов типа родитель-потомок.
* Синхронизация типа старт-финиш может считаться обратным вариантом синхронизации типа финиш-старт.
* Синхронизация типа финиш-финиш. Одна задача не может завершиться до тех пор, пока не завершится другая, т.е. задача A не может финишировать до задачи B.

Существует ряд программных инструментов, которые могут помочь разработчику защитить разделяемые данные и сделать код потокобезопасным, их называют примитивами синхронизации, среди которых наиболее распространенные – мьютексы, семафоры, условные переменные и спин-блокировки [4–7]. Все они защищают часть кода, давая только определенному потоку право получать доступ к данным и блокируя остальные.

**Семафор** – это переменная особого типа, которая может изменяться с положительным или отрицательным приращением, но обращение к переменной в ответственный момент всегда атомарно даже в многопоточных программах. Это означает, что, если два или несколько потоков в программе пытаются изменить значение семафора, система гарантирует, что все операции будут на самом деле выполняться одна за другой.

Семафоры лучше всего представлять себе, как счетчики, управляющие доступом к общим ресурсам. Чаще всего они используются как блокирующий механизм, не позволяющий одному процессу захватить ресурс, пока этим ресурсом пользуется другой.

**Мьютекс** (взаимоисключение, mutex) – примитив синхронизации, устанавливающийся в особое сигнальное состояние, когда не занят каким-либо потоком. Только один поток владеет этим объектом в любой момент времени, отсюда и название таких объектов – одновременный доступ к общему ресурсу исключается.

Задача мьютекса – обеспечить такой механизм, чтобы доступ к объекту в определенное время был только у одного потока. Если поток 1 захватил мьютекс объекта А, остальные потоки не получат к нему доступ, чтобы что-то в нем менять. До тех пор, пока мьютекс объекта А не освободится, остальные потоки будут вынуждены ждать.

**4) Как создать поток?**

Для создания нового потока мы можем создать новый класс, либо наследуя его от класса Thread, либо реализуя в классе интерфейс Runnable.

new Thread("JThread").start();

**5) Инициализация атрибутов потока.**

Для управления потоком класс Thread предоставляет еще ряд методов. Наиболее используемые из них:

* **getName():** возвращает имя потока
* **setName(String name):** устанавливает имя потока
* **getPriority():** возвращает приоритет потока
* **setPriority(int proirity):** устанавливает приоритет потока. Приоритет является одним из ключевых факторов для выбора системой потока из кучи потоков для выполнения. В этот метод в качестве параметра передается числовое значение приоритета - от 1 до 10. По умолчанию главному потоку выставляется средний приоритет - 5.
* **isAlive():** возвращает true, если поток активен
* **isInterrupted():** возвращает true, если поток был прерван
* **join():** ожидает завершение потока
* **run():** определяет точку входа в поток
* **sleep():** приостанавливает поток на заданное количество миллисекунд
* **start():** запускает поток, вызывая его метод run()

**6) Освобождение памяти?**

Вызовите функцию **interrupt()** чтобы удалить объект из памяти.

**7) Область конкуренции.**

Критическая секция (Critical Section) это участок кода, в котором поток (thread) получает доступ к ресурсу (например, переменная), который доступен из других потоков.

Для того, чтобы войти в секцию нужно вызвать функцию EnterCriticalSection(), а после завершения работы LeaveCriticalSection().

**8) Состояние отсоединения**

Если поток создается отделенным (PTHREAD\_CREATE\_DETACHED), его ID потока и другие ресурсы могут многократно использоваться, как только он завершится. Если нет необходимости ожидать в вызывающем потоке завершения нового потока, можно вызвать перед его созданием функцию pthread\_attr\_setdetachstate().

Если поток создается неотделенным (PTHREAD\_CREATE\_JOINABLE), предполагается, что создающий поток будет ожидать его завершения и выполнять в созданном потоке pthread\_join(). Независимо от типа потока, процесс не закончится, пока не завершатся все потоки.

pthread\_attr\_setdetachstate() возвращает 0 после успешного завершения или любое другое значение в случае ошибки.

**9) Завершение потока ?**

Завершение потока

Распространенный способ завершения потока представляет опрос логической переменной. И если она равна, например, false, то поток завершает бесконечный цикл и заканчивает свое выполнение.

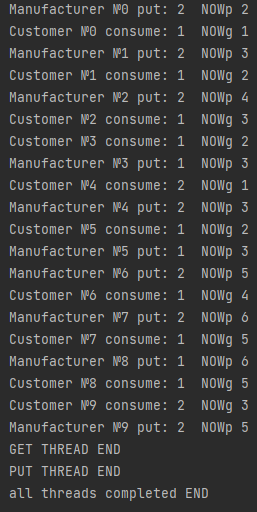
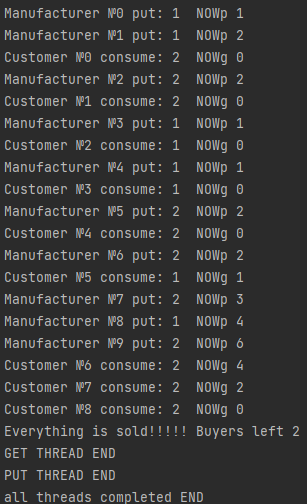
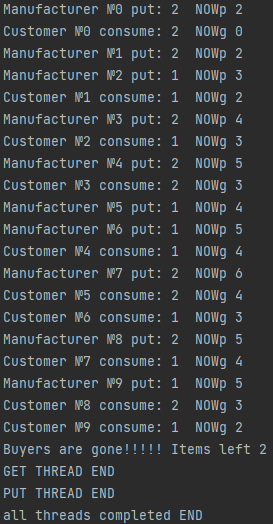
Метод interrupt

Еще один способ вызова завершения или прерывания потока представляет метод interrupt(). Вызов этого метода устанавливает у потока статус, что он прерван. Сам метод возвращает true, если поток может быть прерван, в ином случае возвращается false.

**Вывод**

В данной лабораторной было рассмотрено как создавать потоки и взаимодействовать с ними. Так же были рассмотрены модели параллельного исполнения и основные методы синхронизации действий на основе атомарных операций светофора.

**Результат работы программы**

**  **

**КОД ПРОГРАММЫ:**

package work;  
  
public class Main{  
  
 public static void main(String[] args) throws InterruptedException {  
  
 MyStorage storage = new MyStorage();  
  
 PutThread putThread = new PutThread(storage); //Создание потока  
 GetThread getThread = new GetThread(storage);  
  
 getThread.start(); //Запуск потока  
 putThread.start();  
  
 getThread.join();  
 System.*out*.println("GET THREAD END");  
  
 putThread.join();  
 System.*out*.println("PUT THREAD END");  
  
 System.*out*.println("all threads completed END");  
  
 }  
}

package work;  
  
public class MyStorage {  
 static int *storageAvailable* = 10;  
 static int *takenPlace* = 0;  
  
 public synchronized void get(int power, int personalNumber) {  
 while(*takenPlace* - power < 0){  
 try {  
 wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.print("Customer №" + personalNumber + " consume: " + power);  
 *takenPlace* -= power;  
 System.*out*.println(" NOWg " + *takenPlace*);  
 notify();  
 }  
  
 public synchronized void put(int count, int personalNumber) {  
 while (*takenPlace* + count >= *storageAvailable*) {  
 try {  
 wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.print("Manufacturer №" + personalNumber + " put: " + count);  
 *takenPlace* += count;  
 System.*out*.println(" NOWp " + *takenPlace*);  
 notify();  
 }  
}

package work;  
  
public class GetThread extends Thread  
{  
 MyStorage storage;  
 int power = 0;  
  
 public GetThread(MyStorage storage) {  
 this.storage = storage;  
 }  
  
 @Override  
 public void run() //Этот метод будет выполнен в побочном потоке  
 {  
 int counter=0;  
 for (int i = 0; i < 10; i++){  
 power = (int)(Math.*random*()\*2+1);  
 storage.get(power, counter); counter++;  
 if (PutThread.*iterations* >= 10 && MyStorage.*takenPlace* == 0){  
 System.*out*.println("Everything is sold!!!!! Buyers left " + (10 - i));  
 break;  
 }  
  
 try {  
 *sleep*((int)(Math.*random*() \* 1000));  
 } catch (InterruptedException ignored) { }  
 }  
 if (PutThread.*iterations* >= 10 && MyStorage.*takenPlace* != 0) {  
 System.*out*.println("Buyers are gone!!!!! Items left " + MyStorage.*takenPlace*);  
 }  
 }  
}

package work;  
  
public class PutThread extends Thread {  
  
 static int *iterations* = 0;  
 MyStorage storage;  
 int power = 0;  
  
 public PutThread(MyStorage storage) {  
 this.storage = storage;  
 }  
  
 @Override  
 public void run() //Этот метод будет выполнен в побочном потоке  
 {  
 for (*iterations* = 0; *iterations* < 10; *iterations*++){  
 power = (int)(Math.*random*()\*2+1);  
 storage.put(power, *iterations*);  
 try {  
 *sleep*((int)(Math.*random*() \* 1000));  
 } catch (InterruptedException ignored) { }  
 }  
 }  
}