# ЗАДАНИЕ №2

Для заданной группы вычислительных процессов организовать доступ к критический секции с использованием монитора.

Объяснить достоинства и недостатки каждого из методов взаимного исключения или организации доступа к разделяемым ресурсам. Привести примеры использования в Windows.

В 1974 году Хоар и Бринч Хансен предложили примитив синхронизации более высокого уровня, называемый монитором.

В параллельном программировании монитор – это пассивный набор разделяемых переменных и повторно входимых процедур доступа к ним, которым процессы пользуются в режиме разделения, причем в каждый момент им может пользоваться только один процесс.

Таким образом, монитор – это механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для реализации

динамического распределения конкретного общего ресурса или группы общих

ресурсов. Процесс, желающий получить доступ к разделяемым переменным, должен обратиться к монитору, который либо предоставит доступ, либо откажет в нем. Необходимость входа в монитор с обращением к какой-либо его процедуре (например, с запросом на выделение требуемого ресурса) может

возникать у многих процессов. Однако вход в монитор находится под жестким

контролем – здесь осуществляется взаимоисключение процессов, так что в каждый момент времени только одному процессу разрешается войти в монитор. Процессам, которые хотят войти в монитор, когда он уже занят, приходится ждать, причем режимом ожидания автоматически управляет сам монитор. При отказе в доступе монитор блокирует обратившийся к нему процесс и определяет условие, по которому процесс ждет. Проверка условия выполняется самим монитором, который и деблокирует ожидающий процесс. Поскольку механизм монитора гарантирует взаимоисключение процессов, отсутствуют серьезные проблемы, связанные с организацией параллельных вычислительных процессов.

В вычислительной технике проблема производителя-потребителя (также известная как проблема ограниченного буфера) является классическим примером проблемы многопроцессорной синхронизации. Проблема описывает два процесса, производителя и потребителя, которые совместно используют общий буфер фиксированного размера, используемый в качестве очереди.

* Задача производителя - сгенерировать данные, поместить их в буфер и начать заново.
* В то же время потребитель потребляет данные (т.е. удаляет их из буфера), по одной части за раз.

**Проблема**

Чтобы убедиться, что производитель не будет пытаться добавить данные в буфер, если он заполнен, и что потребитель не будет пытаться удалить данные из пустого буфера.

**Решение**

Производитель должен либо перейти в спящий режим, либо сбросить данные, если буфер заполнен. В следующий раз, когда потребитель удаляет

элемент из буфера, он уведомляет производителя, который снова начинает заполнять буфер. Таким же образом потребитель может перейти в спящий режим, если обнаружит, что буфер пуст. В следующий раз, когда производитель помещает данные в буфер, он пробуждает спящего потребителя.

Неверное решение может привести к тупиковой ситуации, когда оба процесса ожидают пробуждения.

**Важные примечания**

* + В классе PC (класс, который имеет методы как producer, так и consumer), добавляется связанный список заданий и емкость списка, чтобы проверить, что производитель не производит, если список заполнен.
  + В классе Producer значение инициализируется как 0.
    - Также у нас есть бесконечный внешний цикл для вставки значений в список. Внутри этого цикла у нас есть синхронизированный блок, так что одновременно работает только поток производителя или потребителя.
    - Перед добавлением заданий в список существует внутренний цикл, который проверяет, заполнен ли список заданий, поток- производитель отказывается от встроенной блокировки на PC и переходит в состояние ожидания.
    - Если список пуст, элемент управления переходит под цикл и добавляет значение в список.
  + В классе Consumer у нас снова есть бесконечный цикл для извлечения значения из списка.
    - Внутри у нас также есть внутренний цикл, который проверяет, пуст ли список.
    - Если он пуст, мы заставляем потребительский поток снять блокировку на PC и передаем управление потоку producer для создания новых заданий.
    - Если список не пустой, мы обходим цикл и удаляем элемент из списка.
  + В обоих методах мы используем notify в конце всех операторов. Причина проста: если у нас есть что-то в списке, мы можем заставить поток потребителя потреблять это, или если мы что-то потребляли, мы можем заставить производителя производить что-то.
  + Функция sleep () в конце обоих методов просто заставляет вывод программы выполняться пошагово и не отображает все сразу, чтобы мы могли видеть, что на самом деле происходит в программе.

Монитор:

Синхронизация включена:

package com.company;

import java.util.LinkedList; public class Main {

public static void main(String[] args) throws InterruptedException

{

// Объект класса, который имеет оба метода produce()

// и consume()

final PC pc = new PC();

// Создание потока производителя

Thread t1 = new Thread(new Runnable() { @Override

public void run()

{

try { pc.produce();

}

catch (InterruptedException e) { e.printStackTrace();

}

}

});

// Создание потока потребителя

Thread t2 = new Thread(new Runnable() { @Override

public void run()

{

try { pc.consume();

}

catch (InterruptedException e) { e.printStackTrace();

}

}

});

// Запуск двух потоков

t1.start();

t2.start();

// t1 заканчивается до t2 t1.join();

t2.join();

}

// Этот класс имеет список, producer(добавляет элементы в список

// и consumer (удаляет элементы). public static class PC {

// Создание общего списка для производителя и потребителя

// Размер списка - 2.

LinkedList<Integer> list = new LinkedList<>(); int capacity = 2;

// Функция вызывается потоком производителя public void produce() throws InterruptedException

{

int value = 0; while (true) {

synchronized (this)

{

// поток производителя ждет, пока список

// заполнится

while (list.size() == capacity) wait();

System.out.println("Producer produced-"

+ value); list.add(value++);

// уведомляет поток потребителей о том, что

// теперь они могут начать потреблять notify();

Thread.sleep(1000);

}

}

}

// Функция, вызываемая потоком потребителя

public void consume() throws InterruptedException

{

while (true) { synchronized (this)

{

// поток потребителя ждет, пока список

// будет пустой

while (list.size() == 0) wait();

int val = list.removeFirst(); System.out.println("Consumer consumed-"

+ val);

// Пробуждает поток производителя notify();

// и засыпает Thread.sleep(1000);

}

}

}

}

}

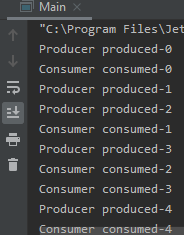


Рисунок – режим работы, когда монитор запущен.

Синхронизация выключена:

public class Main {

public static void main(String[] args) throws InterruptedException

{

// Объект класса, который имеет оба метода produce()

// и consume()

final PC pc = new PC();

// Создание потока производителя Thread t1 = new Thread(new Runnable() {

@Override

public void run()

{

try { pc.produce();

}

catch (InterruptedException e) { e.printStackTrace();

}

}

});

// Создание потока потребителя

Thread t2 = new Thread(new Runnable() { @Override

public void run()

{

try { pc.consume();

}

catch (InterruptedException e) { e.printStackTrace();

}

}

});

// Запуск двух потоков t1.start();

t2.start();

// t1 заканчивается до t2 t1.join();

t2.join();

}

// Этот класс имеет список, producer(добавляет элементы в список

// и consumer (удаляет элементы). public static class PC {

// Создание общего списка для производителя и потребителя

// Размер списка - 2.

LinkedList<Integer> list = new LinkedList<>(); int capacity = 1;

// Функция вызывается потоком производителя public void produce() throws InterruptedException

{

int value = 0; while (true) {

synchronized (this)

{

// поток производителя ждет, пока список

// заполнится

/\* while (list.size() == capacity) wait();

\*/

System.out.println("Producer produced-"

+ value); list.add(value++);

// уведомляет поток потребителей о том, что

// теперь они могут начать потреблять

//notify();

Thread.sleep(1000);

}

}

}

// Функция, вызываемая потоком потребителя

public void consume() throws InterruptedException

{

int valuee = 0; while (true) {

synchronized (this)

{

// поток потребителя ждет, пока список

// будет пустой

/\*while (list.size() == 0) wait();\*/

//int val = list.removeFirst(); System.out.println("Consumer consumed-"

+ valuee); list.add(valuee++);

// Пробуждает поток производителя

//notify();

// и засыпает Thread.sleep(1000);

}

}

}

}

}

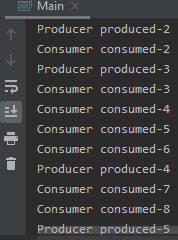


Рисунок – режим работы, когда монитор выключен.