Частное учреждение образования
Колледж бизнеса и права

УТВЕРЖДАЮ
Заведующий
методическим кабинетом
_____Е.В. Паскал
«___»_____2021

Специальность: 2-40 01 01 «Программное обеспечение информационных технологий»	Дисциплина: «Основы кроссплатформенного программирования»

ЛАБОРАТОРНАЯ РАБОТА № 9
Инструкционно-технологическая карта

Тема: «Многопоточные программы на языке Java»

Цель: Научиться работать с потоками, создавать их, решать проблемы многопоточности.

Время выполнения: 4 часа

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретические сведения;
2. Ответить на контрольные вопросы;
3. Откомпилировать примеры программ из раздела «Теоретические сведения»;
4. Выполнить ИДЗ.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ ДЛЯ ВЫПОЛНЕНИЯ РАБОТЫ

Наиболее очевидная область применения многопоточности – это программирование интерфейсов. Многопоточность незаменима тогда, когда необходимо, чтобы графический интерфейс продолжал отзываться на действия пользователя во время выполнения некоторой обработки информации. Например, поток, отвечающий за интерфейс, может ждать завершения другого потока, загружающего файл из интернета, и в это время выводить некоторую анимацию или обновлять прогресс-бар. Кроме того, он может остановить поток загружающий файл, если была нажата кнопка «отмена».

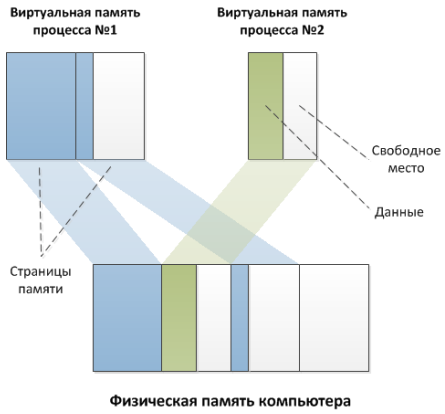
Процессы

Процесс — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Чаще всего одна программа состоит из

одного процесса, но бывают и исключения (например, браузер Chrome создает отдельный процесс для каждой вкладки, что дает ему некоторые преимущества, вроде независимости вкладок друг от друга). Процессы изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен (взаимодействие между процессами осуществляется с помощью специальных средств).

Для каждого процесса ОС создает так называемое «виртуальное адресное пространство», к которому процесс имеет прямой доступ. Это пространство принадлежит процессу, содержит только его данные и находится в полном его распоряжении. Операционная система же отвечает за то, как виртуальное пространство процесса проецируется на физическую память.

Схема этого взаимодействия представлена на картинке. Операционная система оперирует так называемыми страницами памяти, которые представляют собой просто область определенного фиксированного размера. Если процессу становится недостаточно памяти, система выделяет ему дополнительные страницы из физической памяти. Страницы виртуальной памяти могут проецироваться на физическую память в произвольном порядке.



При запуске программы операционная система создает процесс, загружая в его адресное пространство код и данные программы, а затем запускает главный поток созданного процесса.

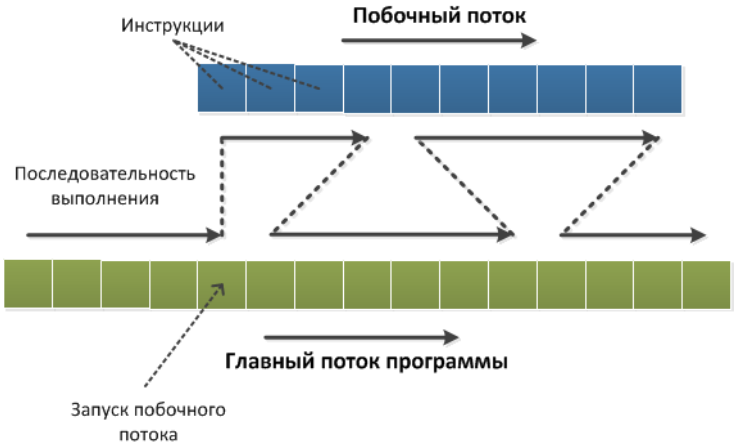
Потоки

Один поток – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.

Следует отдельно обговорить фразу «параллельно с другими потоками». Известно, что на одно ядро процессора, в каждый момент времени, приходится одна единица исполнения. То есть одноядерный процессор может обрабатывать команды только последовательно, по одной за раз (в упрощенном случае). Однако запуск нескольких параллельных потоков возможен и в системах с одноядерными процессорами. В этом случае система будет периодически переключаться между потоками, поочередно давая выполняться то одному, то другому потоку. Такая схема

называется псевдо-параллелизмом. Система запоминает состояние (контекст) каждого потока, перед тем как переключиться на другой поток, и восстанавливает его по возвращению к выполнению потока. В контекст потока входят такие параметры, как стек, набор значений регистров процессора, адрес исполняемой команды и прочее…

Проще говоря, при псевдопараллельном выполнении потоков процессор мечется между выполнением нескольких потоков, выполняя по очереди часть каждого из них.



Цветные квадраты на рисунке – это инструкции процессора (зеленые – инструкции главного потока, синие – побочного). Выполнение идет слева направо. После запуска побочного потока его инструкции начинают выполняться вперемешку с инструкциями главного потока. Кол-во выполняемых инструкций за каждый подход не определено.

Запуск потоков

Каждый процесс имеет хотя бы один выполняющийся поток. Тот поток, с которого начинается выполнение программы, называется главным. В языке Java, после создания процесса, выполнение главного потока начинается с метода main(). Затем, по мере необходимости, в заданных программистом местах, и при выполнении заданных им же условий, запускаются другие, побочные потоки.

В языке Java поток представляется в виде объекта-потомка класса Thread. Этот класс инкапсулирует стандартные механизмы работы с потоком.

Запустить новый поток можно двумя способами:

Способ 1

Создать объект класса Thread, передав ему в конструкторе нечто, реализующее интерфейс Runnable. Этот интерфейс содержит метод run(), который будет выполняться в новом потоке. Поток закончит выполнение, когда завершится его метод run().

Выглядит это так:

```
class SomeThing                    //Нечто, реализующее интерфейс Runnable
implements Runnable          //(содержащее метод run())
{
        public void run()       //Этот метод будет выполняться в побочном потоке
        {
```

```
                System.out.println("Привет из побочного потока!");
        }
    }

    public class Program                    //Класс с методом main()
    {
        static SomeThing mThing;    //mThing - объект класса, реализующего
интерфейс Runnable

        public static void main(String[] args)
        {
            mThing = new SomeThing();

            Thread myThready = new Thread(mThing);//Создание потока
"myThready"
            myThready.start();                          //Запуск потока

            System.out.println("Главный поток завершён...");
        }
    }
```

Для пущего укорочения кода можно передать в конструктор класса Thread объект безымянного внутреннего класса, реализующего интерфейс Runnable:

```
    public class Program{                   //Класс с методом main().
        public static void main(String[] args){        {
            Thread myThready = new Thread(new Runnable();{//Создание потока
                public void run() //Этот метод будет выполняться в побочном
потоке                  {
                        System.out.println("Привет из побочного потока!");
                    }
            });
            myThready.start();    //Запуск потока
            System.out.println("Главный поток завершён...");
        }
    }
```

Завершение потоков

В Java существуют (существовали) средства для принудительного завершения потока. В частности, метод Thread.stop() завершает поток незамедлительно после своего выполнения. Однако этот метод, а также Thread.suspend(), приостанавливающий поток, и Thread.resume(), продолжающий выполнение потока, были объявлены устаревшими и их использование отныне крайне нежелательно. Дело в том, что поток может быть «убит» во время выполнения операции, обрыв которой на полуслове оставит некоторый объект в неправильном состоянии, что приведет к появлению трудноотлавливаемой и случайным образом возникающей ошибке.

Вместо принудительного завершения потока применяется схема, в которой каждый поток сам ответственен за своё завершение. Поток может остановиться либо тогда, когда он закончит выполнение метода run(), (main()

— для главного потока) либо по сигналу из другого потока. Причем как реагировать на такой сигнал — дело, опять же, самого потока. Получив его, поток может выполнить некоторые операции и завершить выполнение, а может и вовсе его проигнорировать и продолжить выполняться. Описание реакции на сигнал завершения потока лежит на плечах программиста.

Java имеет встроенный механизм оповещения потока, который называется Interruption (прерывание, вмешательство), и скоро мы его рассмотрим, но сначала посмотрите на следующую программку:

Incremenator — поток, который каждую секунду прибавляет или вычитает единицу из значения статической переменной Program.mValue. Incremenator содержит два закрытых поля – mIsIncrement и mFinish. То, какое действие выполняется, определяется булевой переменной mIsIncrement — если оно равно true, то выполняется прибавление единицы, иначе — вычитание. А завершение потока происходит, когда значение mFinish становится равно true.

```
class Incremenator extends Thread
{
	//О ключевом слове volatile - чуть ниже
	private volatile boolean mIsIncrement = true;
	private volatile boolean mFinish = false;
	public void changeAction()   //Меняет действие на противоположное
	{
		mIsIncrement = !mIsIncrement;
	}
	public void finish()		//Инициирует завершение потока
	{
		mFinish = true;
	}
	@Override
	public void run()
	{
		do
		{
			if(!mFinish)   //Проверка на необходимость завершения
			{
				if(mIsIncrement)
					Program.mValue++;  //Инкремент
				else
					Program.mValue--;   //Декремент

				//Вывод текущего значения переменной
				System.out.print(Program.mValue + " ");
			}
			else
				return;		//Завершение потока
			try{
				Thread.sleep(1000);   //Приостановка потока на 1 сек.
			}catch(InterruptedException e){}
		}
```

```
                while(true);
        }
}
public class Program
{
        //Переменая, которой оперирует инкрементатор
        public static int mValue = 0;
        static Incremenator mInc;       //Объект побочного потока
        public static void main(String[] args)
        {
                mInc = new Incremenator();  //Создание потока
                System.out.print("Значение = ");
                mInc.start();    //Запуск потока
                //Троекратное изменение действия инкрементатора
                //с интервалом в i*2 секунд
                for(int i = 1; i <= 3; i++)
                {
                        try{
                                Thread.sleep(i*2*1000); //Ожидание в течении i*2 сек.
                        }catch(InterruptedException e){}
                        mInc.changeAction();//Переключение действия
                }
                mInc.finish(); //Инициация завершения побочного потока
        }
}
```

Консоль:

Значение = 1 2 1 0 -1 -2 -1 0 1 2 3 4

Взаимодействовать с потоком можно с помощью метода changeAction() (для смены вычитания на сложение и наоборот) и метода finish() (для завершения потока).

В объявлении переменных mIsIncrement и mFinish было использовано ключевое слово volatile (изменчивый, не постоянный). Его необходимо использовать для переменных, которые используются разными потоками. Это связано с тем, что значение переменной, объявленной без volatile, может кэшироваться отдельно для каждого потока, и значение из этого кэша может различаться для каждого из них. Объявление переменной с ключевым словом volatile отключает для неё такое кэширование и все запросы к переменной будут направляться непосредственно в память.

В этом примере показано, каким образом можно организовать взаимодействие между потоками. Однако есть одна проблема при таком подходе к завершению потока — Incremenator проверяет значение поля mFinish раз в секунду, поэтому может пройти до секунды времени между тем, когда будет выполнен метод finish(), и фактическим завершения потока. Было бы замечательно, если бы при получении сигнала извне, метод sleep() возвращал выполнение и поток незамедлительно начинал своё завершение. Для выполнения такого сценария существует встроенное средство оповещения потока, которое называется Interruption (прерывание, вмешательство).

Interruption

Класс Thread содержит в себе скрытое булево поле, подобное полю mFinish в программе Incremenator, которое называется флагом прерывания. Установить этот флаг можно вызвав метод interrupt() потока. Проверить же, установлен ли этот флаг, можно двумя способами. Первый способ — вызвать метод bool isInterrupted() объекта потока, второй — вызвать статический метод bool Thread.interrupted(). Первый метод возвращает состояние флага прерывания и оставляет этот флаг нетронутым. Второй метод возвращает состояние флага и сбрасывает его. Заметьте что Thread.interrupted() — статический метод класса Thread, и его вызов возвращает значение флага прерывания того потока, из которого он был вызван. Поэтому этот метод вызывается только изнутри потока и позволяет потоку проверить своё состояние прерывания.

Итак, вернемся к нашей программе. Механизм прерывания позволит нам решить проблему с засыпанием потока. У методов, приостанавливающих выполнение потока, таких как sleep(), wait() и join() есть одна особенность — если во время их выполнения будет вызван метод interrupt() этого потока, они, не дожидаясь конца времени ожидания, сгенерируют исключение InterruptedException.

Переделаем программу Incremenator – теперь вместо завершения потока с помощью метода finish() будем использовать стандартный метод interrupt(). А вместо проверки флага mFinish будем вызывать метод bool Thread.interrupted();

Так будет выглядеть класс Incremenator после добавления поддержки прерываний:

```
class Incremenator extends Thread
{
	private volatile boolean mIsIncrement = true;
	public void changeAction()   //Меняет действие на противоположное
	{
		mIsIncrement = !mIsIncrement;
	}
	@Override
	public void run()
	{
		do
		{
			if(!Thread.interrupted())	//Проверка прерывания
			{
				if(mIsIncrement) Program.mValue++;	//Инкремент
				else Program.mValue--;			//Декремент
				//Вывод текущего значения переменной
				System.out.print(Program.mValue + " ");
			}
			else
				return;		//Завершение потока
			try{
```

```
                        Thread.sleep(1000);   //Приостановка потока на 1 сек.
                    }catch(InterruptedException e){
                        return; //Завершение потока после прерывания
                    }
                }
                while(true);
        }
}

class Program
{
        //Переменая, которой оперирует инкрементатор
        public static int mValue = 0;
        static Incremenator mInc;       //Объект побочного потока
        public static void main(String[] args)
        {
                mInc = new Incremenator();  //Создание потока
                System.out.print("Значение = ");
                mInc.start();    //Запуск потока
                //Троекратное изменение действия инкрементатора
                //с интервалом в i*2 секунд
                for(int i = 1; i <= 3; i++)
                {
                        try{
                                Thread.sleep(i*2*1000);//Ожидание в течении i*2 сек.
                        }
                        catch (InterruptedException e){}
                        mInc.changeAction();//Переключение действия
                }
                mInc.interrupt();        //Прерывание побочного потока
        }
}
```

Консоль:

Значение = 1 2 1 0 -1 -2 -1 0 1 2 3 4

Как видите, мы избавились от метода finish() и реализовали тот же механизм завершения потока с помощью встроенной системы прерываний. В этой реализации мы получили одно преимущество — метод sleep() вернет управление (сгенерирует исключение) незамедлительно после прерывания потока.

Заметьте что методы sleep() и join() обёрнуты в конструкции try-catch. Это необходимое условие работы этих методов. Вызывающий их код должен перехватывать исключение InterruptedException, которое они бросают при прерывании во время ожидания.

Метод Thread.sleep()

Thread.sleep() — статический метод класса Thread, который приостанавливает выполнение потока, в котором он был вызван. Во время выполнения метода sleep() система перестает выделять потоку процессорное время, распределяя его между другими потоками. Метод sleep() может выполняться либо заданное кол-во времени (миллисекунды или

наносекунды) либо до тех пор пока он не будет остановлен прерыванием (в этом случае он сгенерирует исключение InterruptedException).

```
Thread.sleep(1500);                 //Ждет полторы секунды
Thread.sleep(2000, 100);            //Ждет 2 секунды и 100 наносекунд
```

Несмотря на то, что метод sleep() может принимать в качестве времени ожидания наносекунды, не стоит принимать это всерьез. Во многих системах время ожидания все равно округляется до миллисекунд а то и до их десятков.

Метод yield()

Статический метод Thread.yield() заставляет процессор переключиться на обработку других потоков системы. Метод может быть полезным, например, когда поток ожидает наступления какого-либо события и необходимо чтобы проверка его наступления происходила как можно чаще. В этом случае можно поместить проверку события и метод Thread.yield() в цикл:

```
//Ожидание поступления сообщения
while(!msgQueue.hasMessages())              //Пока в очереди нет сообщений
{
        Thread.yield();                     //Передать управление другим потокам
}
```

Метод join()

В Java предусмотрен механизм, позволяющий одному потоку ждать завершения выполнения другого. Для этого используется метод join(). Например, чтобы главный поток подождал завершения побочного потока myThready, необходимо выполнить инструкцию myThready.join() в главном потоке. Как только поток myThready завершится, метод join() вернет управление, и главный поток сможет продолжить выполнение.

Метод join() имеет перегруженную версию, которая получает в качестве параметра время ожидания. В этом случае join() возвращает управление либо когда завершится ожидаемый поток, либо когда закончится время ожидания. Подобно методу Thread.sleep() метод join может ждать в течение миллисекунд и наносекунд – аргументы те же.

С помощью задания времени ожидания потока можно, например, выполнять обновление анимированной картинки пока главный (или любой другой) поток ждёт завершения побочного потока, выполняющего ресурсоёмкие операции:

```
Thinker brain = new Thinker();      //Thinker - потомок класса Thread.
brain.start();              //Начать "обдумывание".
do
{
        mThinkIndicator.refresh();  //mThinkIndicator - анимированная картинка.
        try{
                brain.join(250);    //Подождать окончания мысли четверть секунды.
        }catch(InterruptedException e){}
}
while(brain.isAlive());         //Пока brain думает...
//brain закончил думать (звучат овации).
```

В этом примере поток brain (мозг) думает над чем-то, и предполагается, что это занимает у него длительное время. Главный поток ждет его четверть секунды и, в случае, если этого времени на раздумье не хватило, обновляет «индикатор раздумий» (некоторая анимированная картинка). В итоге, во время раздумий, пользователь наблюдает на экране индикатор мыслительного процесса, что дает ему знать, что электронные мозги чем-то заняты.

Приоритеты потоков

Каждый поток в системе имеет свой приоритет. Приоритет – это некоторое число в объекте потока, более высокое значение которого означает больший приоритет. Система в первую очередь выполняет потоки с большим приоритетом, а потоки с меньшим приоритетом получают процессорное время только тогда, когда их более привилегированные собратья простаивают.

Работать с приоритетами потока можно с помощью двух функций:

void setPriority(int priority) – устанавливает приоритет потока.

Возможные значения priority — MIN_PRIORITY, NORM_PRIORITY и MAX_PRIORITY.

int getPriority() – получает приоритет потока.

Некоторые полезные методы класса Thread

boolean isAlive() — возвращает true если myThready() выполняется и false если поток еще не был запущен или был завершен.

setName(String threadName) – Задает имя потока.

String getName() – Получает имя потока.

Имя потока – ассоциированная с ним строка, которая в некоторых случаях помогает понять, какой поток выполняет некоторое действие. Иногда это бывает полезным.

static Thread Thread.currentThread() — статический метод, возвращающий объект потока, в котором он был вызван.

long getId()– возвращает идентификатор потока. Идентификатор – уникальное число, присвоенное потоку.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое потоки?
2. Объясните понятие процесса.
3. Опишите методы создания потоков.
4. Как синхронизировать потоки?
5. В чем главная проблема многопоточности в джава?
6. Как можно обработать взаимодействие с потоком в джава?
7. Опишите приоритеты потоков.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Задание 1.

Выведете состояние потока перед его запуском, после запуска и во время выполнения.

Задание 2.

Напишите программу, в которой создаются два потока, которые выводят на консоль своё имя по очереди.

Задание 3.

Дано два потока — производитель и потребитель. Производитель генерирует некоторые данные (в примере — числа). Потребитель «потребляет» их. Два потока разделяют общий буфер данных, размер которого ограничен. Если буфер пуст, потребитель должен ждать, пока там появятся данные. Если буфер заполнен полностью, производитель должен ждать, пока потребитель заберёт данные и место освободится.

ДОМАШНЕЕ ЗАДАНИЕ

Индивидуальное задание

ЛИТЕРАТУРА

У. Савитч, язык Java. Курс программирования, Вильямс, 2017

Преподаватель А.С.Кибисова

Рассмотрено на заседании цикловой комиссии программного обеспечения информационных технологий
Протокол № ____ от «___» _____ 2021
Председатель ЦК _____В.Ю.Михалевич