Многопоточность в Java

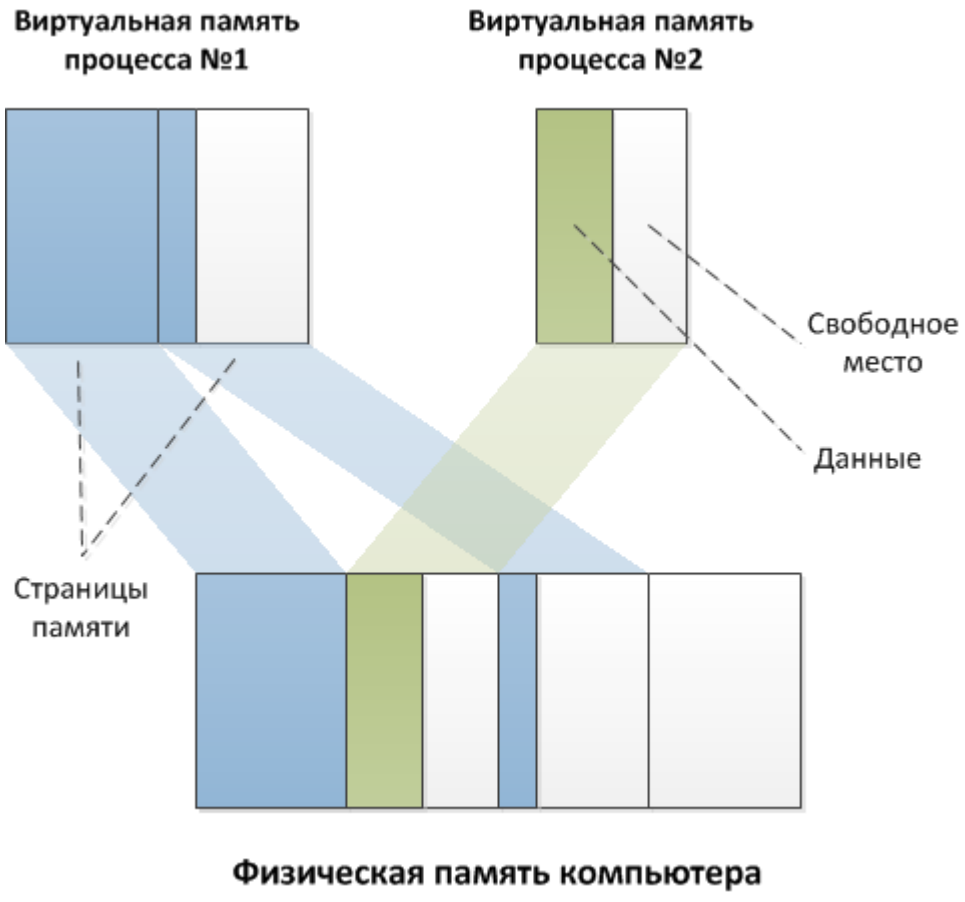
Наиболее очевидная область применения многопоточности – это программирование интерфейсов. Многопоточность незаменима тогда, когда необходимо, чтобы графический интерфейс продолжал отзываться на действия пользователя во время выполнения некоторой обработки информации. Например, поток, отвечающий за интерфейс, 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.

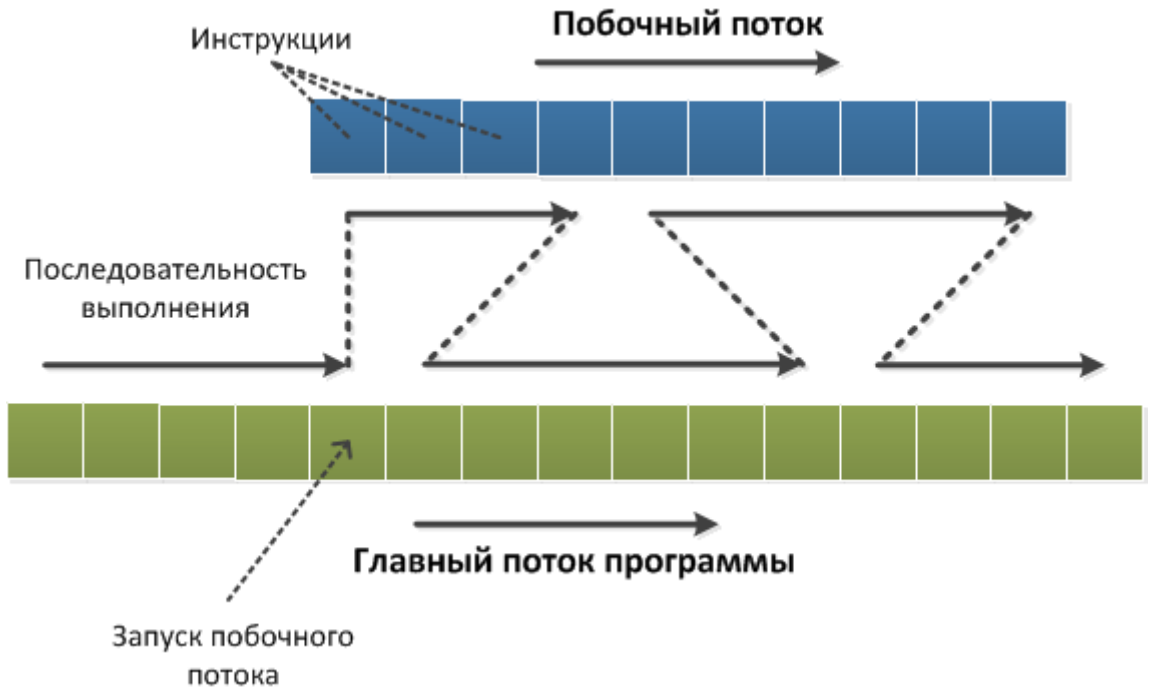
Потоки

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.

Вот как это выглядит:



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.

То, что инструкции параллельных потоков выполняются вперемешку, в некоторых случаях может привести к конфликтам доступа к данным.

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:

1 способ:

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)
      {
            //Создание потока
            Thread myThready = new Thread(new Runnable()
            {
                  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...");
      }
}
```

Поскольку Runnable фактически представляет функциональный интерфейс, который определяет один метод, то объект этого интерфейса мы можем представить в виде лямбда-выражения:

```
public class Program          //Класс с методом main().
{
      public static void main(String[] args)
      {
            Runnable r = () -> {
                  System.out.println("Привет из побочного потока!");
                };
            //Создание потока
            Thread myThready = new Thread(r);
            myThready.start(); //Запуск потока

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

Способ 2

Создать потомка класса Thread и переопределить его метод run():

```
class AffableThread extends Thread
{
      @Override
      public void run() //Этот метод будет выполнен в побочном потоке
      {
            System.out.println("Привет из побочного потока!");
      }
}

public class Program
{
      static AffableThread mSecondThread;

      public static void main(String[] args)
      {
            mSecondThread = new AffableThread(); //Создание потока
            mSecondThread.start();                          //Запуск потока

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

В приведённом выше примере в методе main() создается и запускается еще один поток. Важно отметить, что после вызова метода mSecondThread.start() главный поток продолжает своё выполнение, не дожидаясь пока порожденный им поток завершится. И те инструкции, которые идут после вызова метода start(), будут выполнены параллельно с инструкциями потока mSecondThread.

Асинхронность выполнения кода

Асинхронность означает то, что нельзя утверждать, что какая-либо инструкция одного потока, выполнится раньше или позже инструкции другого. Или, другими словами, параллельные потоки независимы друг от друга,

за исключением тех случаев, когда программист сам описывает зависимости между потоками с помощью предусмотренных для этого средств языка.

Завершение процесса и демоны

В Java процесс завершается тогда, когда завершается последний его поток. Даже если метод main() уже завершился, но еще выполняются порожденные им потоки, система будет ждать их завершения.

Однако это правило не относится к особому виду потоков – демонам. Если завершился последний обычный поток процесса, и остались только потоки-демоны, то они будут принудительно завершены и выполнение процесса закончится. Чаще всего потоки-демоны используются для выполнения фоновых задач, обслуживающих процесс в течение его жизни.

Объявить поток демоном достаточно просто — нужно перед запуском потока вызвать его метод `setDaemon(true)`;
Проверить, является ли поток демоном, можно вызвав его метод `boolean isDaemon()`;

Interruption

Класс Thread содержит в себе скрытое булево поле. 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.

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.

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).

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).
```

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.