# Поток-средсвто помог выполн одновр неск задач

Монитор- механизм взаимод и синхронизации процессов

Процесс-совокупн кода и данных,разделяющих общее вирт адресное про-во

Синхорн- процесс,позволяющий выполн все паралл потоки в проге синхронно.

# Каким образом можно создать поток?

В ранбл созда объекс сред и в него передаём что-то реализующее итнрфейс ранабл

a) 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...");

}

}

b) 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.

# 2. Что такое монитор?

Как известно, каждый объект в java имеет свой **монитор**, и потому, в отличие от того же C++, нет необходимости guard-ить доступ к объектам отдельными mutex-ами. Для достижения эффектов взаимного исключения и синхронизации потоков используют следующие операции:

* **monitorenter**: захват монитора. В один момент времени монитором может владеть лишь один поток. Если на момент попытки захвата монитор занят, поток, пытающийся его захватить, будет ждать до тех пор, пока он не освободится. При этом, потоков в очереди может быть несколько.
* **monitorexit**: освобождение монитора

# 3. Как работают методы wait и notify/notifyAll?

Иногда при взаимодействии потоков встает вопрос о извещении одних потоков о действиях других. Например, действия одного потока зависят от результата действий другого потока, и надо как-то известить один поток, что второй поток произвел некую работу. И для подобных ситуаций у класса Object определено ряд методов:

wait(): освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()

notify(): продолжает работу потока, у которого ранее был вызван метод wait()

notifyAll(): возобновляет работу всех потоков, у которых ранее был вызван метод wait()

Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

# 4. Чем отличается работа метода wait с параметром и без параметра?

Без параметров ждёт notify(). С параметрами тоже ждёт notify, но ещё и время задать можно.

# 5. Как работает метод Thread.yield()? Чем отличаются методы

Thread.sleep() и Thread.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

}

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

**DEADLOCK** Описанные условия блокировки представляют наиболее простой по диагностике случай взаимной блокировки. Зачастую в многопоточных приложениях различные объекты пытаются получить доступ к одним и тем же синхронизированным блокам. При этом может возникнуть взаимная блокировка.

Допустим, один поток исполнения входит в монитор объекта Х, а другой - в мо­нитор объекта У. Если поток исполнения в объекте Х попытается вызвать любой син­хронизированный метод для объекта У, он будет блокирован, как и предполагалось.

Но если поток исполнения в объекте У, в свою очередь, попытается вызвать любой синхронизированный метод для объекта Х, то этот поток будет ожидать вечно, по­скольку для получения доступа к объекту Х он должен снять свою блокировку с объек­та У, чтобы первый поток исполнения мог завершиться.

# 6. Как работает метод Thread.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.

# 7. Что такое dead lock?

# 8. Что значит приоритет потока?

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.

# 9. Что такое потоки - демоны в Java?

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();

# 10. Что значит усыпить поток?

Thread.sleep();

# 11. В каких состояниях может быть поток в Java? Как вообще работает поток?

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.

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.

# 12. Различия между CyclicBarrier и CountDownLatch?

CyclicBarrier (потоки ожидают точки, после чего барьер снимается), CountDownLatch (потоки ожидать завершения заданного числа операций)

# 13. Как работает Semaphor?

Семафоры представляют еще одно средство синхронизации для доступа к ресурсу. В Java семафоры представлены классом Semaphore из пакета java.util.concurrent.

Для управления доступом к ресурсу семафор использует счетчик, представляющий количество разрешений. Если значение счетчика больше нуля, то поток получает доступ к ресурсу, при этом счетчик уменьшается на единицу. После окончания работы с ресурсом поток освобождает семафор, и счетчик увеличивается на единицу. Если же счетчик равен нулю, то поток блокируется и ждет, пока не получит разрешение от семафора.

Установить количество разрешений для доступа к ресурсу можно с помощью конструкторов класса Semaphore:

Semaphore(int permits)

Semaphore(int permits, boolean fair)

Параметр permits указывает на количество допустимых разрешений для доступа к ресурсу. Параметр fair во втором конструкторе позволяет установить очередность получения доступа. Если он равен true, то разрешения будут предоставляться ожидающим потокам в том порядке, в каком они запрашивали доступ. Если же он равен false, то разрешения будут предоставляться в неопределенном порядке.

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: