\*Информация из текста такого цвета не обязательна для ответа, но желательна для понимания, либо пригодится при ответе на дополнительные вопросы. Часть 1 - минимально необходимая информация для защиты 6 и 8 лабораторных. Часть 2 - расширенные данные для понимания устройства потоков, их работы и применении.

**Часть 1**

**Для защиты 6 лабораторной**

**Поток и процесс - что такое поток?**

**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. В это пространство ОС загружает код и данные программы, а затем запускает главный поток созданного процесса. Адресное 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. После создания процесса ОС запускает главный поток созданного процесса.

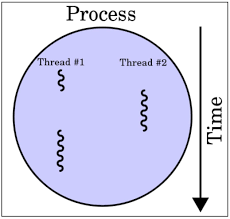
**Поток** – это одна единица исполнения кода или способ выполнения процесса. Каждый поток выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса. Данные из адресной области процесса доступны всем его потокам.

**Для понимания**

Потоки не параллельны:)

Как правило выполняется **псевдопараллельное** выполнение потоков - это когда процессор мечется между выполнением нескольких потоков, выполняя по очереди часть каждого из них. Такой подход позволяет реализовать многопоточность на одноядерном процессоре, который имеет один физический поток

Вот как выглядит работа выполнение процесса 2 потоками:



## **Создание потока - как можно создать поток в java?**

1. Создать потомка класса **Thread** и переопределить его метод **run()**;
2. Создать объект класса **Thread**, передав ему в конструкторе экземпляр класса, реализующего интерфейс **Runnable**. Эти интерфейс содержит метод **run()**, который будет выполняться в новом потоке. Поток закончит выполнение, когда завершится его метод **run()**.
3. Создать потомка класса **SwingWorker** (именно так в 6 лабораторной создаются потоки) и переопределить его методы **doInBackground()** и **done()**
4. Вызвать метод submit() у экземпляра класса реализующего интерфейс ExecutorService, передав ему в качестве параметра экземпляр класса реализующего интерфейс Runnable или Callable (содержит метод call(), в котором описывается логика выполнения).

Пример:

класс потомка Thread:

public class SomeThreadClass extends Thread {

private int threadName;

public int getThreadName(){

return this.threadName;

}

public SomeThreadClass(int num){

this.threadName = num;

}

public SomeThreadClass(){

this.threadName = -1;

}

@Override

public void run(){

System.out.println("I'm Thread number: " + this.threadName);

}

}

класс, реализующий интерфейс Runnable:

public class IImplemetsRunnable implements Runnable{

private int threadName;

public IImplemetsRunnable(int num){

this.threadName = num;

}

public IImplemetsRunnable(){

this.threadName = -1;

}

@Override

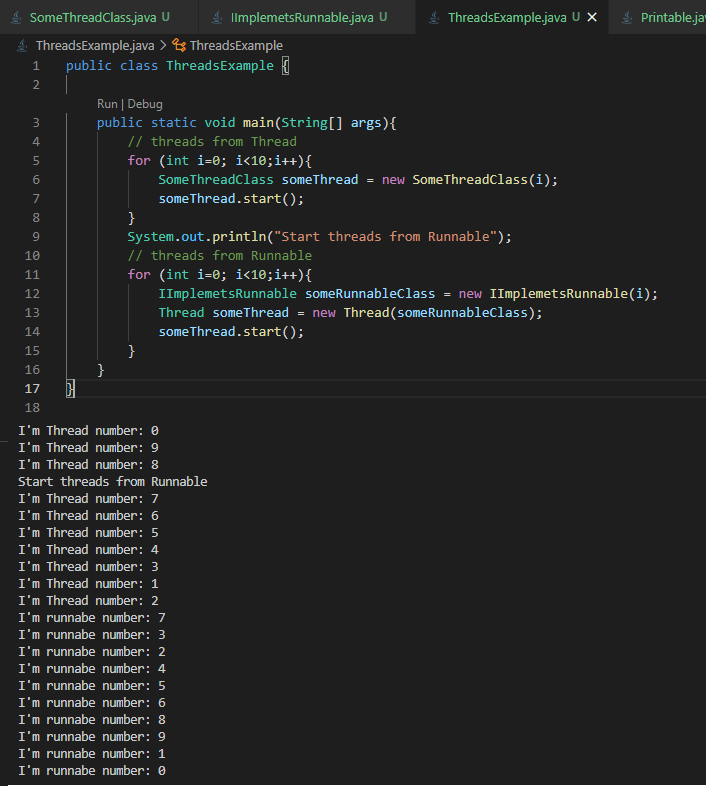
public void run(){

System.out.println("I'm runnabe number: " + this.threadName);

}

}

Использование экземпляров классов потоков:



## **Чем различаются Thread и Runnable?**

Thread - это класс, содержит в себе функционал для управления потоком.

Runnable - это интерфейс, представляющий абстракцию над выполняемой задачей.

Помимо того, что Runnable помогает разрешить проблему множественного наследования, несомненный плюс от его использования состоит в том, что он позволяет логически отделить логику выполнения задачи от непосредственного управления потоком.

## **В чём заключается разница между методами start() и run()?**

Несмотря на то, что start() вызывает метод run() внутри себя, это не то же самое, что просто вызов run(). Если run() вызывается как обычный метод, то он вызывается в том же потоке и никакой новый поток не запускается, как это происходит, в случае, когда вы вызываете метод start().

## **В каких состояниях может находиться поток?**

Потоки могут находиться в одном из следующих состояний:

* **Новый (New)**. После создания экземпляра потока, он находится в состоянии Новый до тех пор, пока не вызван метод start(). В этом состоянии поток не считается живым.
* **Работоспособный (Runnable)**. Готов выполнить свой метод run() Когда поток находится в этом состоянии, он считается живым. Поток переходит в состояние Работоспособный, когда вызывается метод start(). Поток может перейти в это состояние также из состояния Работающий или из состояния Блокирован.
* **Работающий (Running)**. Поток переходит из состояния Работоспособный в состояние Работающий, когда Планировщик потоков выбирает его как работающий в данный момент.
* **Живой, но не работоспособный** (Alive, but not runnable). Поток может быть живым, но не работоспособным по нескольким причинам:
  + Ожидание (Waiting). Поток переходит в состояние Ожидания, вызывая метод wait(). Вызов notify() или notifyAll() может перевести поток из состояния Ожидания в состояние Работоспособный.
  + Сон (Sleeping). Метод sleep() переводит поток в состояние Сна на заданный промежуток времени в миллисекундах.
  + Блокировка (Blocked). Поток может перейти в это состояние, в ожидании ресурса, такого как ввод/вывод или из-за блокировки другого объекта. В этом случае поток переходит в состояние Работоспособный, когда ресурс становится доступен.
* **Мертвый (Dead)**. Поток считается мертвым, когда его метод run() полностью выполнен. Мертвый поток не может перейти ни в какое другое состояние, даже если для него вызван метод start(). **Отработавший поток нельзя запустить заново!**

**Управление потоком - как завершить поток?**

## **Как принудительно запустить поток?** Никак. В Java не существует абсолютно никакого способа принудительного запуска потока. Это контролируется JVM и Java не предоставляет никакого API для управления этим процессом.

## **Как принудительно завершить поток?** Для прерывания потока можно использовать метод класса Thread - interrupt(), однако это лишь “просьба” о завершении, и в некоторых случаях поток не будет прерван. Для завершения потока можно использовать метод stop(), который не рекомендуется применять, из-за угрозы возникновения блокировок.

**Для защиты 8 лабораторной, но могут и на 6 спросить**

**Синхронизация + Управление потоком - как устроена работа потоков с одним ресурсом?**

Синхронизация - это процесс, который позволяет выполнять потоки параллельно (псевдопараллельнно). Потоки хранят информацию о переменных во временных хранилищах (кэш процессорные регистры и тд), а не непосредственно в общей памяти программы. Из-за чего при использовании одного ресурса несколькими потоками необходимо синхронизировать их работу. В java для этого используются **synchronized** (см 2 часть) блоки.

## **Методы wait() и notify()/notifyAll()** Эти методы определены у класса Object и предназначены для взаимодействия потоков между собой при межпоточной синхронизации.

* wait(): освобождает монитор(контроль за ресурсом) и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()/notifyAll();
* notify(): продолжает работу потока, у которого ранее был вызван метод wait();
* notifyAll(): возобновляет работу всех потоков, у которых ранее был вызван метод wait().

## **Методы Thread.sleep() и Thread.yield()** Метод yield() служит причиной того, что поток переходит из состояния работающий (running) в состояние работоспособный (runnable), давая возможность другим потокам активизироваться. Но следующий выбранный для запуска поток может и не быть другим.

Метод sleep() вызывает засыпание текущего потока на заданное время, состояние изменяется с работающий (running) на ожидающий (waiting).

***«Усыпить»* поток** значит приостановить его на определенный промежуток времени, вызвав в ходе его выполнения статический метод Thread.sleep() передав в качестве параметра необходимое количество времени в миллисекундах. До истечения этого времени поток может быть выведен из состояния ожидания вызовом interrupt() с выбрасыванием InterruptedException.

**Вопрос на понимание:**

## Какое из следующих утверждений о потоках неверно?

1. Если метод start() вызывается дважды для одного и того же объекта Thread, во время выполнения генерируется исключение.
2. Порядок, в котором запускались потоки, может не совпадать с порядком их фактического выполнения.
3. Если метод run() вызывается напрямую для объекта Thread, во время выполнения генерируется исключение.
4. Если метод sleep() вызывается для потока, во время выполнения синхронизированного кода, блокировка не снимается.

Правильный ответ: 3. Если метод run() вызывается напрямую для объекта Thread, во время выполнения исключение не генерируется. Однако, код, написанный в методе run() будет выполняться текущим, а не новым потоком. Таким образом, правильный способ запустить поток – это вызов метода start(), который приводит к выполнению метода run() новым потоком.

Вызов метода start() дважды для одного и того же объекта Thread приведёт к генерированию исключения IllegalThreadStateException во время выполнения, следовательно, утверждение 1 верно. Утверждение 2 верно, так как порядок, в котором выполняются потоки, определяется Планировщиком потоков, независимо от того, какой поток запущен первым. Утверждение 4 верно, так как поток не освободит блокировки, которые он держит, когда он переходит в состояние Ожидания.

**Часть 2**

**Теория работы потоков в java**

*Видимость (visibility)*. Один поток может в какой-то момент временно сохранить значение некоторых полей не в основную память, а в регистры или локальный кэш процессора, таким образом второй поток, выполняемый на другом процессоре, читая из основной памяти, может не увидеть последних изменений поля. И наоборот, если поток на протяжении какого-то времени работает с регистрами и локальными кэшами, читая данные оттуда, он может сразу не увидеть изменений, сделанных другим потоком в основную память.

К вопросу видимости имеют отношение следующие ключевые слов языка Java: synchronized, volatile, final.

С точки зрения Java все переменные (за исключением локальных переменных, объявленных внутри метода) хранятся в главной памяти, которая доступна всем потокам, кроме этого, каждый поток имеет локальную—рабочую—память, где он хранит копии переменных, с которыми он работает, и при выполнении программы поток работает только с этими копиями. Надо отметить, что это описание не требование к реализации, а всего лишь модель, которая объясняет поведение программы, так, в качестве локальной памяти не обязательно выступает кэш память, это могут быть регистры процессора или потоки могут вообще не иметь локальной памяти.

При входе в synchronized метод или блок поток обновляет содержимое локальной памяти, а при выходе из synchronized метода или блока поток записывает изменения, сделанные в локальной памяти, в главную. Такое поведение synchronized методов и блоков следует из правил для отношения «происходит раньше»: так как все операции с памятью происходят раньше освобождения монитора и освобождение монитора происходит раньше захвата монитора, то все операции с памятью, которые были сделаны потоком до выхода из synchronized блока должны быть видны любому потоку, который входит в synchronized блок для того же самого монитора. Очень важно, что это правило работает только в том случае, если потоки синхронизируются, используя один и тот же монитор

Также модель памяти определяет дополнительную семантику ключевого слова final, имеющую отношение к видимости: после того как объект был корректно создан, любой поток может видеть значения его final полей без дополнительной синхронизации. «Корректно создан» означает, что ссылка на создающийся объект не должна использоваться до тех пор, пока не завершился конструктор объекта. Наличие такой семантики для ключевого слова final позволяет создание неизменяемых (immutable) объектов, содержащих только final поля, такие объекты могут свободно передаваться между потоками без обеспечения синхронизации при передаче.

Есть одна проблема, связанная с final полями: реализация разрешает менять значения таких полей после создания объекта (это может быть сделано, например, с использованием механизма reflection). Если значение final поля—константа, чьё значение известно на момент компиляции, изменения такого поля могут не иметь эффекта, так-как обращения к этой переменной могли быть заменены компилятором на константу. Также спецификация разрешает другие оптимизации, связанные с final полями, например, операции чтения final переменной могут быть переупорядочены с операциями, которые потенциально могут изменить такую переменную. Так что рекомендуется изменять final поля объекта только внутри конструктора, в противном случае поведение не специфицировано.

*Reordering (переупорядочивание)*. Для увеличения производительности процессор/компилятор могут переставлять местами некоторые инструкции/операции. Вернее, с точки зрения потока, наблюдающего за выполнением операций в другом потоке, операции могут быть выполнены не в том порядке, в котором они идут в исходном коде. Тот же эффект может наблюдаться, когда один поток кладет результаты первой операции в регистр или локальный кэш, а результат второй операции попадает непосредственно в основную память. Тогда второй поток, обращаясь к основной памяти может сначала увидеть результат второй операции, и только потом первой, когда все регистры или кэши синхронизируются с основной памятью. Еще одна причина reordering, может заключаться в том, что процессор может решить поменять порядок выполнения операций, если, например, сочтет что такая последовательность выполнится быстрее.

Монитор, мьютекс (mutex) – это средство обеспечения контроля за доступом к ресурсу. У монитора может быть максимум один владелец в каждый текущий момент времени. Следовательно, если кто-то использует ресурс и захватил монитор для обеспечения единоличного доступа, то другой, желающий использовать тот же ресурс, должен подождать освобождения монитора, захватить его и только потом начать использовать ресурс.

Удобно представлять монитор как id захватившего его объекта. Если этот id равен 0 – ресурс свободен. Если не 0 – ресурс занят. Можно встать в очередь и ждать его освобождения.

В Java у каждого экземпляра объекта есть монитор, который контролируется непосредственно виртуальной машиной. Используется он так: любой нестатический synchronized-метод при своем вызове прежде всего пытается захватить монитор того объекта, у которого он вызван (на который он может сослаться как на this). Если это удалось – метод исполняется. Если нет – поток останавливается и ждет, пока монитор будет отпущен.

В Java все объекты имеют одну блокировку, благодаря которой только один поток одновременно может получить доступ к критическому коду в объекте. Такая синхронизация помогает предотвратить повреждение состояния объекта. Если поток получил блокировку, ни один другой поток не может войти в синхронизированный код, пока блокировка не будет снята. Когда поток, владеющий блокировкой, выходит из синхронизированного кода, блокировка снимается. Теперь другой поток может получить блокировку объекта и выполнить синхронизированный код. Если поток пытается получить блокировку объекта, когда другой поток владеет блокировкой, поток переходит в состояние Блокировки до тех пор, пока блокировка не снимется.

**Потокобезопасность** – свойство объекта или кода, которое гарантирует, что при исполнении или использовании несколькими потоками, код будет вести себя, как предполагается. Например потокобезопасный счётчик не пропустит ни один счёт, даже если один и тот же экземпляр этого счётчика будет использоваться несколькими потоками.

## **Потоки-демоны**

Потоки-демоны работают в фоновом режиме вместе с программой, но не являются неотъемлемой частью программы. Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения и его деятельность заключается в обслуживании основных потоков приложения, то такой процесс может быть запущен как поток-демон с помощью метода setDaemon(boolean value), вызванного у потока до его запуска. Метод boolean isDaemon() позволяет определить, является ли указанный поток демоном или нет. Базовое свойство потоков-демонов заключается в возможности основного потока приложения завершить выполнение потока-демона (в отличие от обычных потоков) с окончанием кода метода main(), не обращая внимания на то, что поток-демон еще работает.

Основной поток программы нельзя сделать демоном. Потоки-демоны позволяют описывать фоновые процессы, которые нужны только для обслуживания основных потоков выполнения и не могут существовать без них.

**Создание потока**

## **Чем отличаются два интерфейса Runnable и Callable?**

* Интерфейс Runnable появился в Java 1.0, а интерфейс Callable был введен в Java 5.0 в составе библиотеки java.util.concurrent;
* Классы, реализующие интерфейс Runnable для выполнения задачи должны реализовывать метод run(). Классы, реализующие интерфейс Callable - метод call();
* Метод Runnable.run() не возвращает никакого значения, Callable.call() возвращает объект Future, который может содержать результат вычислений;
* Метод run() не может выбрасывать проверяемые исключения, в то время как метод call() может.

**Управление потоком**

## **Остановка потока**

На данный момент в Java принят уведомительный порядок остановки потока (хотя JDK 1.0 и имеет несколько управляющих выполнением потока методов, например stop(), suspend() и resume() - в следующих версиях JDK все они были помечены как deprecated из-за потенциальных угроз взаимной блокировки).

Для корректной остановки потока можно использовать метод класса Thread - interrupt(). Этот метод выставляет некоторый внутренний флаг-статус прерывания. В дальнейшем состояние этого флага можно проверить с помощью метода isInterrupted() или Thread.interrupted() (для текущего потока). Метод interrupt() также способен вывести поток из состояния ожидания или спячки. Т.е. если у потока были вызваны методы sleep() или wait() – текущее состояние прервется и будет выброшено исключение InterruptedException. Флаг в этом случае не выставляется.

Схема действия при этом получается следующей:

* Реализовать поток.
* В потоке периодически проводить проверку статуса прерывания через вызов isInterrupted().
* Если состояние флага изменилось или было выброшено исключение во время ожидания/спячки, следовательно поток пытаются остановить извне.
* Принять решение – продолжить работу (если по каким-то причинам остановиться невозможно) или освободить заблокированные потоком ресурсы и закончить выполнение.

Возможная проблема, которая присутствует в этом подходе – блокировки на потоковом вводе-выводе. Если поток заблокирован на чтении данных - вызов interrupt() из этого состояния его не выведет. Решения тут различаются в зависимости от типа источника данных. Если чтение идет из файла – долговременная блокировка крайне маловероятна и тогда можно просто дождаться выхода из метода read(). Если же чтение каким-то образом связано с сетью – стоит использовать неблокирующий ввод-вывод из Java NIO.

Второй вариант реализации метода остановки (а также и приостановки) – сделать собственный аналог interrupt(). Т.е. объявить в классе потока флаги – на остановку и/или приостановку и выставлять их путем вызова заранее определённых методов извне. Методика действия при этом остаётся прежней – проверять установку флагов и принимать решения при их изменении. Недостатки такого подхода. Во-первых, потоки в состоянии ожидания таким способом не «оживить». Во-вторых, выставление флага одним потоком совсем не означает, что второй поток тут же его увидит. Для увеличения производительности виртуальная машина использует кэш данных потока, в результате чего обновление переменной у второго потока может произойти через неопределенный промежуток времени (хотя допустимым решением будет объявить переменную-флаг как volatile).

## **Почему не рекомендуется использовать метод Thread.stop()?**

При принудительной остановке (приостановке) потока, stop() прерывает поток в недетерменированном месте выполнения, в результате становится совершенно непонятно, что делать с принадлежащими ему ресурсами. Поток может открыть сетевое соединение - что в таком случае делать с данными, которые еще не вычитаны? Где гарантия, что после дальнейшего запуска потока (в случае приостановки) он сможет их дочитать? Если поток блокировал разделяемый ресурс, то как снять эту блокировку и не приведет ли принудительное снятие к нарушению консистентности системы? То же самое можно расширить и на случай соединения с базой данных: если поток остановят посередине транзакции, то кто ее будет закрывать? Кто и как будет разблокировать ресурсы?

**Синхронизация**

## **Какие существуют способы синхронизации в Java?**

* Системная синхронизация с использованием wait()/notify(). Поток, который ждет выполнения каких-либо условий, вызывает у этого объекта метод wait(), предварительно захватив его монитор. На этом его работа приостанавливается. Другой поток может вызвать на этом же самом объекте метод notify() (опять же, предварительно захватив монитор объекта), в результате чего, ждущий на объекте поток «просыпается» и продолжает свое выполнение. В обоих случаях монитор надо захватывать в явном виде, через synchronized-блок, потому как методы wait()/notify() не синхронизированы!
* Системная синхронизация с использованием join(). Метод join(), вызванный у экземпляра класса Thread, позволяет текущему потоку остановиться до того момента, как поток, связанный с этим экземпляром, закончит работу.
* Использование классов из пакета java.util.concurrent, который предоставляет набор классов для организации межпоточного взаимодействия. Примеры таких классов - Lock, Semaphore и пр.. Концепция данного подхода заключается в использовании атомарных операций и переменных.

Когда вызван метод wait(), поток освобождает блокировку на объекте и переходит из состояния Работающий (Running) в состояние Ожидания (Waiting). Метод notify() подаёт сигнал одному из потоков, ожидающих на объекте, чтобы перейти в состояние Работоспособный (Runnable). При этом невозможно определить, какой из ожидающих потоков должен стать работоспособным. Метод notifyAll() заставляет все ожидающие потоки для объекта вернуться в состояние Работоспособный (Runnable). Если ни один поток не находится в ожидании на методе wait(), то при вызове notify() или notifyAll() ничего не происходит.

Поток может вызвать методы wait() или notify() для определённого объекта, только если он в данный момент имеет блокировку на этот объект. wait(), notify() и notifyAll() должны вызываться только из синхронизированного кода.

**Виды блокировок потока**

## **Deadlock**

Взаимная блокировка (**deadlock**) - явление, при котором все потоки находятся в режиме ожидания. Происходит, когда достигаются состояния:

1. взаимного исключения: по крайней мере один ресурс занят в режиме неделимости и, следовательно, только один поток может использовать ресурс в любой данный момент времени.
2. удержания и ожидания: поток удерживает как минимум один ресурс и запрашивает дополнительные ресурсов, которые удерживаются другими потоками.
3. отсутствия предочистки: операционная система не переназначивает ресурсы: если они уже заняты, они должны отдаваться удерживающим потокам сразу же.
4. цикличного ожидания: поток ждёт освобождения ресурса, другим потоком, который в свою очередь ждёт освобождения ресурса заблокированного первым потоком.

Простейший способ избежать взаимной блокировки – не допускать цикличного ожидания. Этого можно достичь, получая мониторы разделяемых ресурсов в определённом порядке и освобождая их в обратном порядке.

## **Livelock**

**livelock** – тип взаимной блокировки, при котором несколько потоков выполняют бесполезную работу, попадая в зацикленность при попытке получения каких-либо ресурсов. При этом их состояния постоянно изменяются в зависимости друг от друга. Фактической ошибки не возникает, но КПД системы падает до 0. Часто возникает в результате попыток предотвращения deadlock.

Реальный пример livelock, – когда два человека встречаются в узком коридоре и каждый, пытаясь быть вежливым, отходит в сторону, и так они бесконечно двигаются из стороны в сторону, абсолютно не продвигаясь в нужном им направлении

synchronized - это зарезервированное слово позволяет добиваться синхронизации в помеченных им методах или блоках кода.

## **Race condition**

Состояние гонки (race condition) - ошибка проектирования многопоточной системы или приложения, при которой эта работа напрямую зависит от того, в каком порядке выполняются потоки. Состояние гонки возникает, когда поток, который должен исполнится в начале, проиграл гонку и первым исполняется другой поток: поведение кода изменяется, из-за чего возникают недетерменированные ошибки.

## **Существует ли способ решения проблемы *race condition*?**

Распространённые способы решения:

* Использование локальной копии — копирование разделяемой переменной в локальную переменную потока. Этот способ работает только тогда, когда переменная одна и копирование производится атомарно (за одну машинную команду), использование volatile.
* Синхронизация - операции над разделяемым ресурсом происходят в синхронизированном блоке (при использовании ключевого слова synchronized).
* Комбинирование методов - вышеперечисленные способы можно комбинировать, копируя «опасные» переменные в синхронизированном блоке. С одной стороны, это снимает ограничение на атомарность, с другой — позволяет избавиться от слишком больших синхронизированных блоков.

Очевидных способов выявления и исправления состояний гонки не существует. Лучший способ избавиться от гонок — правильное проектирование многозадачной системы.