# Процесс

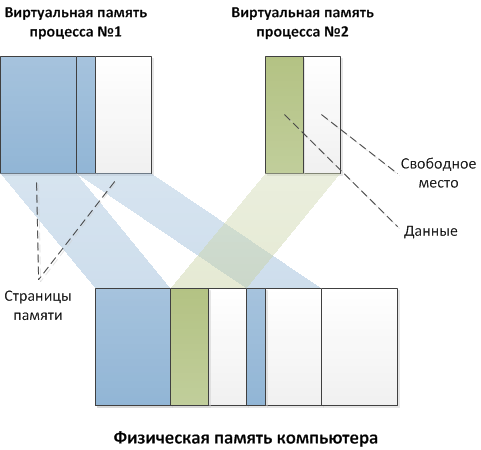
С точки зрения многопоточности, процесс - это экземпляр выполняющейся программы. Внутри процесса могут выполняться один или более потоков. Потоки - это независимые последовательности инструкций, которые могут выполняться параллельно или конкурентно друг с другом.

**Процесс** — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Чаще всего одна программа состоит из одного процесса, но бывают и исключения (браузеры).

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, которой операционная система выделяет процессорное время. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.

В системах с одноядерным процессором, система будет периодически переключаться между потоками, поочередно давая выполняться то одному, то другому потоку. 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.

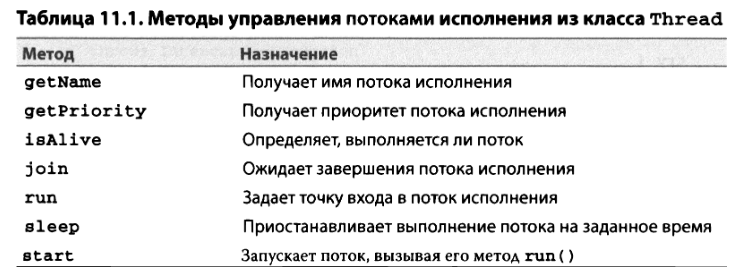
Связь между процессами, и переключение контекста ограничено и обходится дорого. Переключение между потоками дешевая, как и переключение контекста.

Потоки используются для обеспечения асинхронной работы всей исполняющей среды.

# 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, запускаются другие, побочные потоки.

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



*Главный поток* создается автоматически, но им также можно управлять через объект класса Thread. Вызвав метод *currentThread()*, можно получить ссылку на тот поток, из которого он был вызван.

Запустить новый поток можно несколькими 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. В run можно вызвать любые другие методы, объявлять переменные, как и в любом обычном методе. Когда метод run завершается, завершается и сам поток.

Поток запускается, при вызове метода *start()* класса Thread.

**Способ 2**

Расширить класс Thread, а затем получить экземпляр этого класса. В расширяющем классе должен быть объявлен метод *run()*, который является точкой входа в новый поток исполнения. Запуск потока также осуществляется методом start().

Второй способ следует применять, только если мы хотим переопределить какие-то методы класса Thread. В противном случае, лучше реализовывать интерфейс Runnable. Это также позволит нам наследоваться от каких-то других классов.

**Потоки следует запускать именно методом start()!** Если запускать через run, они будут выполняться последовательно в одном потоке.

**3 И 4 СПОСОБ К ВОПРОСУ J22 (Java Concurrency)**

**Способ 3**

Если нужно вернуть значение из потока, то можно реализовывать интерфейс *Callable*. Интерфейс имеет единственный абстрактный метод call, который возвращает значение с помощью объекта Future. Вызвав у future метод get можно получить значение, которое вернул поток. Создаем объект future, в него передаем *Callable*. Затем в конструктор Thread передаем Future и запускаем методом start поток.

Thread thread1 = new Thread(new FutureTask<Integer>(new Call(c)));

**Способ 4**

Если потоков очень много, то удобно использовать *ExecutorService*. В метод submit можно передать Runnable или Callable.

ExecutorService service = Executors.*newFixedThreadPool*(10);  
for (int i=0; i < 100; i++) {  
 service.submit(new Call(c));  
}

Чтобы определить, был ли другой поток завершен, есть 2 метода:

1. Вызвать для потока метод **isAlive()**, который возвращает true, если поток, для которого он был вызван, еще исполняется.
2. Метод **join()**, ожидает завершение того потока, для которого он был вызван. Можно указать промежуток времени, в течении которого требуется ожидать завершения потока.

**Приоритеты потоков**

Каждому потоку исполнения в java присваивается свой приоритет, который определяет поведение данного потока по отношению к другим потокам. Он определяется целым числом. Это число служит для принятия решения о переходе от одного потока к другому.

Чтобы установить приоритет потока, следует вызвать метод **setPriority()**



Уровни приоритета располагаются в промежутке от 1 (MIN\_PRIORITY) до 10(MAX\_PRIORITY). Значение по умолчанию 5(NORM\_PRIORITY).

Метод **GetPriority()** позволяет получить приоритет потока.

# **Синхронизация**

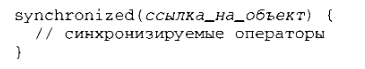
Если несколько потоков используют общие ресурсы, их нужно синхронизировать.

Монитор – объект, используемый в качестве взаимоисключающей блокировки. Только один поток исполнения может в одно и то же время владеть монитором. Можно рассматривать как контейнер, одновременно хранящий только один поток исполнения. Как только один поток войдет в монитор, другие потоки должны дождаться, пока он покинет монитор.

Для синхронизации кода используется ключевое слово **synchronized**.

У объектов имеются свои, неявно связанные с ними мониторы. Чтобы войти в монитор объекта, достаточно вызвать метод, объявленный с модификатором synchronized. Все другие потоки, пытающиеся войти в заблокированный монитор, будут приостановлены до тех пор, пока первый поток не выйдет из монитора. Чтобы выйти из монитора, владелец монитора просто возвращает управление из синхронизированного метода.

Синхронизировать можно не только метод, но и просто блок кода:



В скобках передается ссылка на синхронизируемый объект. Вызов метода, являющегося членом того же класса, на который указывает ссылка, произойдет только тогда, когда текущий поток исполнения войдет в монитор данного объекта.

*Когда вызываем синхронизированный блок, у этого объекта нельзя будет вызвать никакой другой синхронизированный блок. Но несинхронизированный можно.*

Для организации взаимодействия потоков используются методы:

* *wait()* вынуждает вызывающий поток исполнения уступить монитор и перейти в состояние ожидания до тех пор, пока какой-нибуть другой поток не войдет в тот же монитор и не вызовет метод *notify().*
* *notify()* возобновляет исполнение потока, из которого был вызван метод *wait()* для того же самого объекта.
* *notifyAll()* возобновляет исполнение всех потоков, из которых был вызван метод *wait()* для того самого объекта. Одному из этих потоков предоставляется доступ.

Все эти методы могут быть вызваны только из синхронизированного контекста. Они работают на уровне монитора, а монитор назначается объекту, а не конкретному потоку. Поэтому эти методы объявлены в классе Object, а не в Thread. В противном случае, каждый поток должен был бы знать статус другого потока, и не смог узнать что другой поток ожидает доступа к другому ресурсу.

Поток уведомляет другие потоки, что ресурс свободен, через сам ресурс.

Для синхронизации между потоками в Java вы можете использовать ключевое слово `synchronized`, которое позволяет управлять доступом к общим ресурсам. Вы также можете использовать методы `wait`, `notify` и `notifyAll` для организации коммуникации и синхронизации между потоками. Метод `Thread.sleep` используется для приостановки выполнения потока на определенный период времени.

# Мьютекс, семафор, Монитор

Семафоры, мьютексы и мониторы - это различные механизмы синхронизации, используемые в многопоточных программах для обеспечения доступа к общим ресурсам и предотвращения состояний гонки. Вот подробнее о каждом из них:

1. Семафоры (semaphores): Семафоры - это счетчики, которые позволяют контролировать доступ к общим ресурсам для нескольких потоков. Они могут быть использованы для ограничения количества потоков, которые могут получить доступ к определенному ресурсу одновременно. Семафоры могут быть двух типов: бинарные (только два значения: 0 и 1) и счетные (счетчик может быть больше одного). Операции с семафорами включают увеличение счетчика (с помощью операции `V`) и уменьшение счетчика (с помощью операции `P`). Если счетчик становится отрицательным, поток будет заблокирован и ожидать, пока счетчик не станет положительным.

2. Мьютексы (mutexes): Мьютексы - это объекты синхронизации, которые обычно используются для обеспечения взаимного исключения. Они позволяют только одному потоку получить доступ к защищенному ресурсу или критической секции кода в определенный момент времени. Мьютексы могут быть заблокированы (locked) или разблокированы (unlocked). Поток, пытающийся получить доступ к заблокированному мьютексу, будет заблокирован до тех пор, пока мьютекс не будет разблокирован другим потоком.

3. Мониторы (monitors): Мониторы - это абстрактные объекты, которые объединяют данные и код, работающий с этими данными в единую единицу. Мониторы обеспечивают синхронизацию доступа к данным и обеспечивают безопасность потоков. Они предоставляют механизмы для блокировки и разблокировки, чтобы гарантировать, что только один поток может выполнять код внутри монитора в определенный момент времени. Мониторы также могут предоставлять другие операции, такие как ожидание (wait) и оповещение (notify), которые позволяют потокам синхронизироваться и сигнализировать друг другу о состоянии ресурсов.

Мониторы являются более высокоуровневыми и абстрактными, чем семафоры и мьютексы, и часто предоставляются встроенными в язык или библиотеку программирования. Семафоры и мьютексы, с другой стороны, являются более низкоуровневыми и обычно предоставляются операционной системой или низкоуровневыми библиотеками.

Выбор между семафорами, мьютексами и мониторами зависит от конкретной задачи и требований программы. Каждый из этих

**Мьютекс** – специальный объект для синхронизации потоков. Обеспечивает взаимное исключение выполнения участков кодов. Имеет 2 состояния: свободен и занят.

Когда поток получает доступ к объекту, он переводит его в состояния занят. Другие потоки при этом будут ждать освобождения.

В java нет прямого доступа к мьютексу. Доступ к мьютексу есть только у java-машины.

**Монитор –** дополнительная надстройка над мьютексом. Монитор создает защитный механизм. Он срабатывает, когда поток входит в synchronized блок. Монитор в java по сути выражен с помощью слова synchronized.

**Семафор –** механизм синхронизации, который использует счетчик. Счетчик показывает, сколько потоков одновременно могут получать доступ к общему ресурсу.

В java семафоры представлены классом *Semaphore(int permits, Boolean fair)*. Первый параметр указывает кол-во потоков, которые могуn одновременно захватить семафор. *fair* – позволяет установить очередность получения доступа. Если true, то разрешения будут предоставляться потокам в порядке, в котором они запрашивали доступ. Если нет, то в неопределенном порядке.

Метод *acquire()* – увеличить счетчик семафора, *release()* – освободить. Особенность семафора – release может выполнять любой поток, даже тот, который не делал acquire().

Ключевое слово **volatile** указывает компилятору, что все присвоения этой переменной и все операции чтения должны быть атомарными. Например запись long и float на некоторых архитектурах может не быть атомарной, а выполняться в две операции записи по 32 бита.

Присвоение значения volatile переменной имеет связь *happens-before* для последующих чтений из этих переменных. То есть использование volatile обеспечивает нам то, что потоки будут обращаться к актуальному значению переменной. Так же volatile запрещает потокам кэшировать значение для этой переменной.

# **Типы ошибок синхронизации**

**Взаимная блокировка**

Проблема, когда потоки исполнения имеют циклическую зависимость от пары синхронизированных объектов. То есть, когда один поток ждет пока второй поток освободит ресурс. А второй, в свою очередь, ждет пока первый поток освободит ресурс. Потоков может быть и больше (допустим второй блокирует третий, а третий первый).

**Состояние гонки (Race condition)**

Ошибка программирования многозадачной системы, при которой работа системы зависит от того, в каком порядке выполняются потоки. Приведем пример. Пусть, один поток выполняет над общей переменной x операцию x = x + 3, а второй поток - операцию x = x + 5. Данные операции для каждого потока фактически разбиваются на три отдельных подоперации: считать x из памяти, увеличить x, записать x в память. В зависимости от взаимного порядка выполнения потоками подопераций финальное значение переменной x может быть больше исходного на 3, 5 или 8.

**Голодание потоков —** это ситуация, в которой поток не может получить доступ к общим ресурсам, потому что на эти ресурсы всегда претендуют какие-то другие потоки, которым отдаётся предпочтение.

Гонка ресурсов - это ситуация, когда несколько потоков одновременно пытаются получить доступ к общему ресурсу и могут повлиять на его состояние. В Java проблему гонки ресурсов можно решить с помощью синхронизации, используя ключевое слово `synchronized` или другие механизмы синхронизации, такие как мьютексы или мониторы.  
  
Голодание - это ситуация, когда один или несколько потоков не получают достаточно ресурсов для выполнения своих задач. Это может произойти, например, когда потоки ждут доступа к общему ресурсу, и другие потоки постоянно занимают этот ресурс. В Java проблему голодания можно решить с помощью механизмов синхронизации, таких как семафоры или мониторы.  
  
Deadlock (взаимная блокировка) - это ситуация, когда два или более потоков блокируются и ожидают ресурсов, которые удерживают другие потоки. В результате ни один из потоков не может продолжить свое выполнение. В Java проблему взаимной блокировки можно решить, используя правильную упорядоченность получения ресурсов или применяя методы `wait`, `notify` и `notifyAll` для управления состоянием потоков.

**Принципы happens before**

Гласят что какие-то события, связанные с многопоточностью, гарантированно происходят раньше других событий.

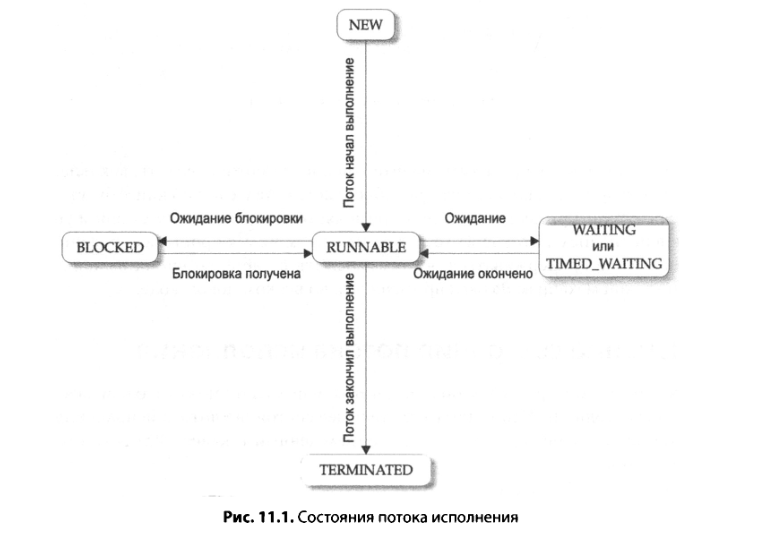
1. Освобождение мьютекса происходит раньше захвата этого же монитора другим потоком.
2. Метод Thread.start() вызывается раньше метода Thread.run().
3. Метод run() завершается раньше выхода из метода join(). Значит, что метод run() отработает до самого конца, и все изменения будут видны в другом потоке, когда он дождется завершения.
4. Запись в переменную volatile происходит раньше чтения из этой же переменной.

**Состояния потока**

Поток исполнения может находится в нескольких состояниях. Чтобы получить состояние потока можно вызвать метод *getState()*. Это метод возвращает значения типа enum Thread.State.

Состояния потока:

* **BLOCKED –** поток приостановил выполнение, поскольку ожидает получения блокировки.
* **NEW –** поток создан, но не начал свое выполнение.
* **RUNNABLE –** Поток выполняется. Переходит в это состояние после вызова метода run().
* **TERMINATED –** поток завершил выполнение.
* **TIMED\_WAITING –** поток приостановил выполнение на определенный промежуток времени, например после вызова метода sleep(). Поток переходит в это состояние и при вызове метода wait() или join().
* **WAITING –** Поток приостановил выполнение, поскольку он ожидает некоторого действия, например после вызова версии метода wait() или join() без заданного времени ожидания.



**ThreadLocal**

Класс, который позволяет хранить переменные, которые должны быть доступны для всего потока. Они так же будут уникальными для каждого потока.

Имеет методы *get()* и *set().*

ThreadLocal гарантирует лишь то, что каждый поток получит ссылку на свой объект, но не изолирует сами объекты. Если 2 разных потока положат в ThreadLocal один и тот же объект, то при доступе к нему будут возникать все проблемы многопоточности.

# **Потокобезопасные concurrent коллекции**

Есть класс Collections в котором есть методы обертки, которые позволяют оборачивать коллекции, делая их синхронизированными:

SynchronizedCollection, synchronizedList, synchronizedMap, synchronizedSet, synchronizedSortedMap, synchronizedSortedSet.

Объектом синхронизации в данном случае является переданный объект. То есть при вызове любого метода потоком будет блокироваться вся коллекция.

Но итератор не является синхронизированным. И при использовании итератора будет вылетать ConcurrentModificatedException, при добавлении элемента во время работы итератора.

Пакет **java.util.concurrent** включает несколько потокобезопасных коллекций:

|  |  |
| --- | --- |
| • [ConcurrentHashMap](https://java-online.ru/concurrent-collections.xhtml#concurrenthashmap) | коллекция типа HashMap, реализующая интерфейс ConcurrentMap; |
| • [CopyOnWriteArrayList](https://java-online.ru/concurrent-collections.xhtml#copyonwritearraylist) | коллекция типа ArrayList с алгоритмом CopyOnWrite; |
| • [CopyOnWriteArraySet](https://java-online.ru/concurrent-collections.xhtml#copyonwritearrayset) | реализация интерфейса Set, использующая за основу CopyOnWriteArrayList; |
| • ConcurrentNavigableMap | расширяет интерфейс NavigableMap; |
| • ConcurrentSkipListMap | аналог коллекции TreeMap с сортировкой данных по ключу и с поддержкой многопоточности; |
| • ConcurrentSkipListSet | реализация интерфейса Set, выполненная на основе класса ConcurrentSkipListMap. |

Класс **ConcurrentHashMap** реализует интерфейс ConcurrentMap, появился в JDK 1.5, является потокобезопасной реализацией Map.

Элементы Entry (пара ключ-значение объявлены как volatile)

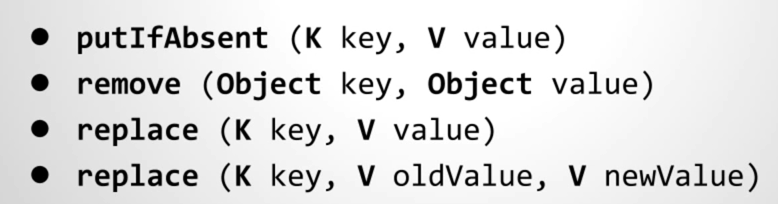
Map использует несколько сегментов, и данный класс нужно рассматривать как группу HashMap’ов. Количество сегментов по умолчанию равно 16.

Если пара ключ-значение хранится в 10-м сегменте, то ConcurrentHashMap заблокирует, при необходимости, только 10-й сегмент, и не будет блокировать остальные 15.

Каждый сегмент представляет собой потокобезопасную таблицу элементов карты. Между хэш-кодами ключей и соответствующими им сегментами устанавливается зависимость на основе применения к старшим разрядам хэш-кода битовой маски.  
В ней так же можно указать предполагаемое количество потоков, которые будут работать с данной коллекцией. Но очень долго расширяется, при перехэшировании. Рекомендуется использовать при частых операциях вставках.

**ConcurrentSkipListMap.** Неблокируещая отсортированная реализация мапы. Реализован в виде skipList (LinkedList, но у него есть указатель на следующий элемент, и на элемент через определенный промежуток (допустим 2 элемент хранит ссылки на 3, 5, 10). Добавление и удаление происходит достаточно быстро, переназначить ссылки). Блокировки локальные, только на перезаписываемые ссылки.

Обе мапы реализуют интерфейс **ConcurrentMap.** В нем есть атомарные операции изменения с проверкой, существует ли элемент.



**ConcurreSkipListSet.** Реализована по тому же принципу что и ConcurrentSkipListMap, т.к. использует ее под капотом.

**CopyOnWriteArrayList.** Массив, хранящий значения помечен volatile. (То есть потоки не будут кэшировать, и всегда будут видеть последние изменения). При чтении не происходит блокировок. При изменении, блокируем коллекцию, создаем копию массива, и затем обновляем ссылку на скопированный массив. Эффективен, если операции чтения частые, а изменения нет.

**CopyOnWriteArraySet.** Аналогичен в работе с CopyOnWriteArrayList.

Concurrent-коллекции в Java предоставляют дополнительные возможности для работы с многопоточностью, которые отсутствуют в классических коллекциях, таких как `Vector` и `Hashtable`. Вот несколько причин, почему вы можете использовать concurrent-коллекции:  
  
1. Потокобезопасность: Concurrent-коллекции предназначены для безопасного использования в многопоточных средах. Они обеспечивают внутреннюю синхронизацию, чтобы гарантировать, что операции чтения и записи будут атомарными и потокобезопасными. Это позволяет избежать проблем гонок ресурсов и взаимной блокировки.  
  
2. Высокая производительность: Concurrent-коллекции разработаны для обеспечения высокой производительности в многопоточных приложениях. Они оптимизированы для эффективного распределения работы между потоками и минимизации блокировок. Некоторые concurrent-коллекции, такие как `ConcurrentHashMap`, могут обрабатывать параллельные операции чтения и записи без блокировки всей коллекции.  
  
3. Итераторы безопасны для использования: Concurrent-коллекции предоставляют итераторы, которые безопасно могут использоваться в многопоточных средах. Итераторы concurrent-коллекций не вызывают ошибок `ConcurrentModificationException`, даже если коллекция изменяется во время итерации.  
  
4. Дополнительные функциональные возможности: Concurrent-коллекции предлагают дополнительные функциональные возможности, которых нет в классических коллекциях. Например, `ConcurrentHashMap` предоставляет атомарные операции, такие как `putIfAbsent`, `remove` и `replace`, которые могут быть безопасно использованы в многопоточных средах.