МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ  
федеральное государственное бюджетное образовательное учреждение высшего образования   
**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**  
**(ФГБОУ ВО «КубГУ»)**

**Кафедра вычислительных технологий**

**Курсовая работа**

**Изучение организации многопоточности в Java и Python**

Работу выполнил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Н.Е.Касаткин  
 (подпись, дата)

Факультет компьютерных технологий и прикладной математики курс 3  
Направление 02.03.02 – «Фундаментальная информатика и информационные технологии»

Научный руководитель к.т.н., доцент\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Т.А. Приходько  
 (подпись, дата)

Нормоконтролер \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   
 (подпись, дата)

Краснодар 2018

Содержание

[ВВЕДЕНИЕ 3](#_Toc532374085)

[1. Основные определения и понятия, принятые в параллельном и распределенном моделировании 4](#_Toc532374086)

[1.1. Определения 4](#_Toc532374087)

[1.2. Понятия многопоточности, многозадачности и многопроцессорности 7](#_Toc532374088)

[2. Средства реализации многопоточности в Python 9](#_Toc532374089)

[2.1. Threading 10](#_Toc532374090)

[2.2. Синхронизация 12](#_Toc532374091)

[3. Многопоточность в Java 13](#_Toc532374092)

[3.1. Способы создания потоков в Java 13](#_Toc532374093)

[3.2. Жизненный цикл потоков 14](#_Toc532374094)

[3.3. Приоритеты потоков 15](#_Toc532374095)

[3.4. Синхронизация 16](#_Toc532374096)

ВВЕДЕНИЕ

(здесь обычно пишут об актуальности темы и о постановке задачи)

1. Основные определения и понятия, принятые в параллельном и распределенном моделировании
   1. Определения

**Многопоточность** – тип многозадачности, в котором все задачи исходят из одной и той же программы (процесса) и, казалось бы, выполняются одновременно. Эта одновременная обработка обманчива, поскольку центральный процессор очень быстро переключает свою активность между всеми потоками (которые являются исполняемыми объектами, каждый со своими стеками, регистрами и счетчиками команд).

Потоки и процессы взаимосвязанные определения, но их следует различать.

**Процесс** – это совокупность кода и данных, имеющих общее виртуальное адресное пространство. Чаще всего программа состоит из одного процесса. Процессы изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен. Если процессу нужно получить доступ к ресурсам другого процесса, необходимо использовать межпроцессное взаимодействие: конвейеры, файлы, каналы связи между компьютерами и многое другое.

**Поток** – набор кода, который служит дополнительной структурой компоновки в программе. Потоки позволяют различным частям программы работать независимо (по времени) друг от друга, а не в фиксированной последовательности и на отдельных процессорах или ядрах одного процессора. Запущенная программа (процесс) может иметь несколько активных потоков одновременно: один может решать математическое уравнение, другой может рисовать график, а третий вести диалог с пользователем. Операционная система решает, какой поток или процесс виден пользователю, а какой остается скрытым.

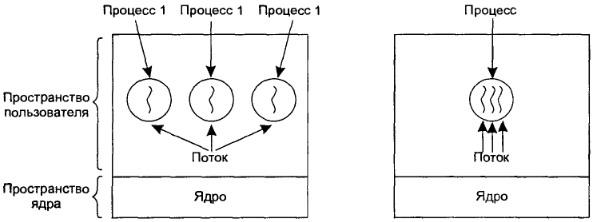


Рисунок 1 – Разница процесса и потока

Следует отдельно пояснить понятие многопоточности на одноядерных процессорах. По определению одноядерный процессор может обрабатывать команды только последовательно. Однако существует технология «псевдо-параллелизма». Смысл данной технологии в том, что система будет время от времени переключаться между потоками, поочерёдно выполняя их. (см. рисунок 2) Для решения некоторых задач это технология даёт значительный выигрыш по времени.

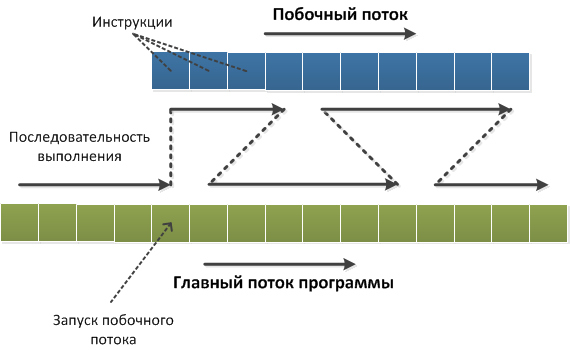


Рисунок 2 – Псевдо-параллелизм

**Синхронизация потоков** - это процесс, который позволяет координировать выполнение всех запущенных потоков, то есть выполнение по определённым правилам, которые устанавливаются программистом.

Для чего это нужно?

Рассмотрим простой пример – операции с банковскими счетами: есть два потока – A и B. Они имеют одинаковую исполняемую часть – снять 10 000 рублей с банковского счёта. Каждый поток знает, что если на счету недостаточно денег, то он не может произвести операцию.

Задача: выполнить потоки A и B. На банковском счету лежит 15 000 рублей.

Рассмотрим выполнение в последовательное (однопоточном) режиме. Сначала выполнится поток A: на счету 15 000 рублей, значит снять 10 000 возможно, следовательно, после выполнения этого потока на счету остаётся 5 000 рублей. После завершения потока A выполняется поток B: при попытке снять 10 000 рублей возникает ошибка – недостаточно средств на счету и поток B завершается. В результате поток А списал 10 000 рублей, поток B ничего не списал и на счету осталось 5 000 рублей.

Теперь же рассмотрим выполнение этой же задачи в многопоточном режиме. Потоки A и B одновременно обращаются к банковскому счёту, смотрят что на нём достаточно денег для снятия и производят списание. В результате поток A списал 10 000 рублей, поток B списал 10 000 рублей и на счету осталось 5 000 рублей. Из доступных 15 000 мы сняли 20 000 и ещё 5 000 осталось на счету. А всё из-за отсутствия синхронизации потоков.

Способы реализации синхронизации для разных языков программирования разные, так что рассмотрим их позже для Java и Python.

* 1. Понятия многопоточности, многозадачности и многопроцессорности

Следует различать такие понятия как:

* Многозадачность: возможность одновременного выполнения нескольких задач (браузер, текстовый редактор, музыкальный проигрыватель).
* Многопоточность: процесс одновременного выполнения нескольких потоков. Многопоточность также называют многозадачностью на основе потоков.
* Многопроцессорность: это то же самое, что и многозадачность, однако при многопроцессорной обработке задействовано более одного процессора в то время как в многозадачности может быть задействован один процессор.
* Параллельная обработка: использование нескольких процессоров в одной компьютерной системе.

Рассмотрим понятие многопоточности.

В современных программах постоянно используется многопоточность, если бы это было не так, то все операции, которые совершал бы пользователь занимали бы гораздо больше времени.

Например, у нас есть сервер, который принимает заявки, обрабатывает их и отправляет ответ. Не будем учитывать время принятия заявки и отправки ответа. Пусть время обработки заявки составляет 5 секунд. Тогда при последовательной работе нашего сервера время обработки 10 заявок составляло бы 50 секунд. Однако, если мы будем использовать многопоточность, то каждую заявку может обрабатывать новый поток (обычно поток обрабатывается ядром процессора, а если все ядра заняты, то создаётся очередь). Время обработки заявок в этом случае (при одном четырёхядерном процессоре) составит около 15 секунд, что более чем в три раза быстрее последовательной версии.

Многопоточность используется для эффективного использования вычислительных ресурсов компьютера. Ведь в последовательной реализации программы присутствует один (главный) поток, а значит все остальные процессоры (ядра этих процессоров) находятся в бездействии (относительно данного процесса).

Одним из главных объектов применения являются компьютерные игры. Так происходит из-за того, что объём данных для обработки постоянно растёт и как бы быстро не развивалось «компьютерное железо» приходится всё равно оптимизировать код так, чтобы он потреблял максимум вычислительных ресурсов.

У многопоточности есть свои недостатки:

* Усложнение кода и его читабельности
* Усложнение процесса отладки
* Увеличение количества скрытых ошибок, которые не всегда выявляются при отладке
* Снижение предсказуемости в работе программы, так как порядок выполнения некоторых операций заранее не известен и программа на разных машинах в разных условиях работает по-разному.

Программист должен понимать, когда нужно реализовывать многопоточность, а когда она только навредит.

1. Средства реализации многопоточности в Python

Организации многопоточности в Python без внешних библиотек можно добиться с помощью модуля *threading.* Однако многопоточности в её привычном понимании в CPython (одна из самых популярных реализаций Python) нет.

Всё дело в так называемом глобальном шлюзе - Global Interpreter Lock (GIL). Этот шлюз ограничивает поведение потоков на уровне интерпретатора. Цель данной технологии – синхронизация потоков, то есть защита от одновременного обращения разных потоков к одному и тому же участку памяти. Благодаря GIL однопоточные приложения работают быстро, а потоки в многопоточных приложениях не конфликтуют. Ценой этому является замедление многопоточных программ.

В один момент времени всегда выполняется только один поток. GIL постоянно (через некоторые промежутки времени – зависит от версии) переключает потоки.

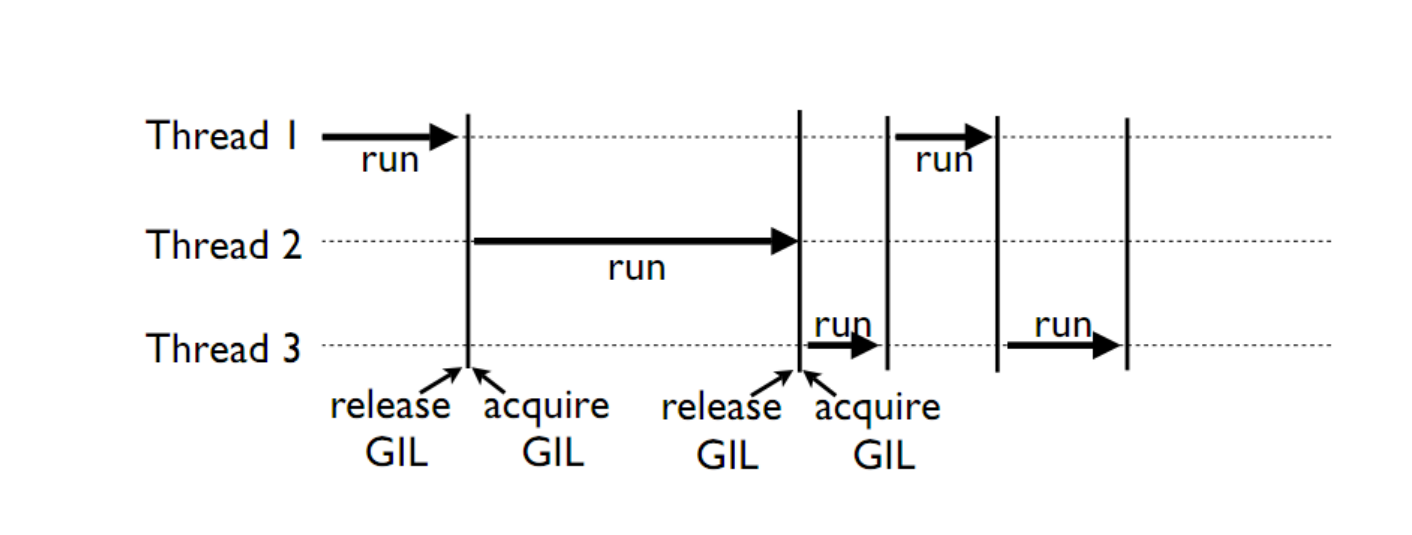


Рисунок 3 – Схема работы GIL

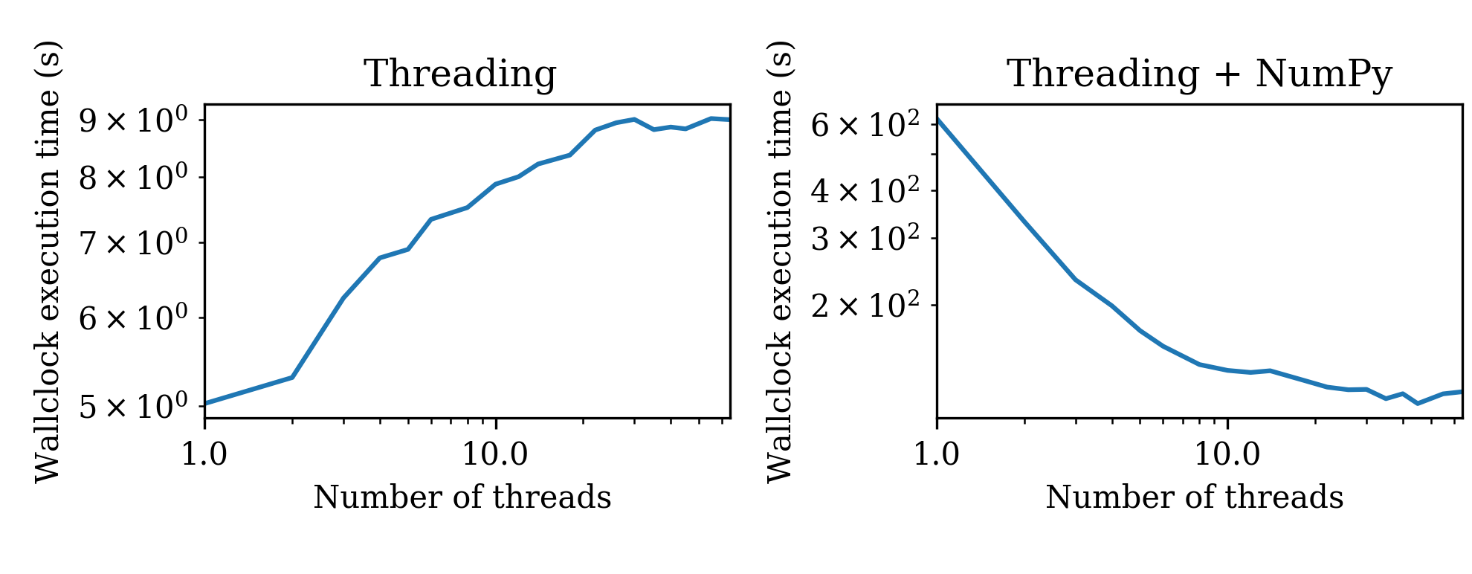
Однако такую работу шлюза можно обойти. Существуют специальные библиотеки, которые помогают ускорять многопоточные программ: NumPy (для математических расчётов), Numba (графические вычисления), SciPy и другие. 

Рисунок 4 – Сравнение времени работы программы с подключением NumPy и без

Таким образом следует понимать, когда стоит использовать многопоточность в Python (когда количество её достоинств будет перекрывать недостатки).

* Для операций ввода-вывода.
* Когда вычисления занимают более миллисекунды.
* Число потоков не превышает количество ядер.
  1. Threading

В данном модуле есть два варианта создания потоков:

1) При помощи вызова функции - threading.Thread(target=, args=)

2) При помощи вызова класса – MyThread(args).start()

Создание класса является более надёжным способом взаимодействия с потоками, к тому же с помощью этого класса легче находить ошибки в отладке программы.

Класс должен иметь метод run() и \_\_init\_\_. В методе run() прописывается все действия, которые должен исполнить каждый поток этого класса. Метод start() запускает экземпляр класса MyThread (см. рисунок 5).

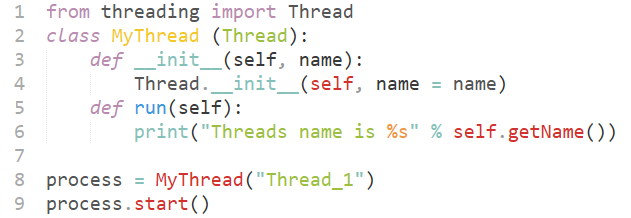


Рисунок 5 – Пример программы с классом

Чтобы управлять потоками, нужно следить, как они себя ведут. В threading для этого есть специальные методы:

**current\_thread()** — возвращает информацию о потоке (имя и его уникальный идентификатор), который вызвал функцию.

**active\_count()** — возвращает количество активных потоков.

**enumerate()** — возвращает список активных потоков.

Можно управлять потоком через методы класса:

**is\_alive()**— возвращает true, если поток активен, иначе false

**getName()** — возвращает имя потока;

**setName(any\_name)** — задаём имя потоку

Обычно Python-приложение не завершается, пока работает хоть один его поток. Но есть особые потоки, которые не мешают закрытию программы и останавливается вместе с ней. Их называют демонами (daemons). Проверить, является ли поток демоном, можно методом **isDaemon()**. Если является, метод вернёт истину.

Назначить поток демоном можно при создании — через параметр “daemon=True” или аргумент в инициализаторе класса.

thread0 **=** Thread(target**=**target\_func, kwargs**=**{‘x’:10}, daemon**=True**)

Можно демонизировать и уже существующий поток методом **setDaemon(daemonic)**.

* 1. Синхронизация
* Атомарные операции: простейший способ синхронизации с общими переменными или другими ресурсами. Атомарная операция – это операция, которая осуществляет только один этап выполнения задачи. Примеры атомарных операций:
* модификация списка на месте (добавление объекта при помощи append);
* выборка объекта в словаре;
* модификация словаря на месте (добавление объекта или вызов метода clear)
* Lock: блокировка, имеет два метода acquire() – осуществляет (захватывает) блокировку, release() – снимает блокировку.
* RLock: блокировка с возможностью повторного захвата, то есть если поток пытается захватить блокировку, которой он уже владеет, то выполнение программы производится в обычном режиме.
* Условные переменные (condition/conditional variable): изменение статуса состояния в приложении (этот статус могут ожидать другие потоки) или оповещение других потоков о наступлении данного состояния.
* Semaphore: механизм синхронизации, который содержит счётчик потоков, которые в данный момент выполняют данный участок кода, если значение счётчика будет 0 (при входе каждого потока в этот участок счётчик декрементируется, при выходе инкрементируется), то поток, который пытается получить доступ будет заблокирован.

1. Многопоточность в Java

Java это язык, поддерживающий многопоточное программирование. По определению многозадачность – это использование общих ресурсов обработки, таких как ЦП, несколькими процессами. Многопоточность расширяет идею многозадачности там, где можно разделить определенные операции в рамках одного приложения на отдельные потоки. Каждый из потоков может выполняться параллельно. Операционная система делит время обработки не только между различными приложениями, но и между каждым потоком в приложении.

* 1. Способы создания потоков в Java

1. Реализуя Runnable интерфейс

Необходимо создать новый класс (implements Runnable) и реализовать метод run(), предоставляемый интерфейсом Runnable. Создать экземпляр объекта Thread, используя конструктор. Вызвать метод start() для экземпляра.

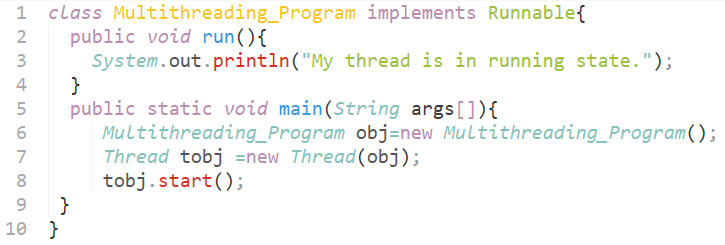


Рисунок 6 – Пример создания потока (implements Runnable)

1. Расширением класса Thread

Необходимо создать новый класс (extends Thread) и переопределить метод run(). Создать экземпляр объекта и вызвать метод start().

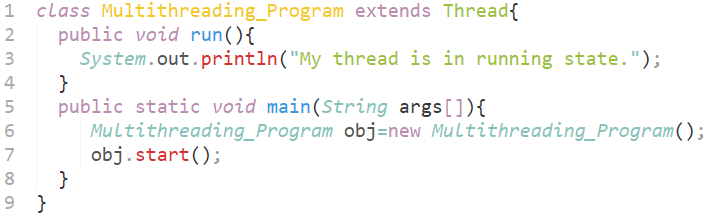


Рисунок 7 – Пример создания потока (extend Threads)

Особенности представленных способов:

Runnable: можно наследовать класс отличный от Thread, Runnable класс нужно передавать в конструктор Thread объекта.

Thread: содержит методы управления потоком.

То есть выбор реализации потоков зависит от метода решения поставленной задачи.

* 1. Жизненный цикл потоков

Возможные состояния потоков:

* **New** – процесс создания потока. Поток находится в этом состоянии пока операционная система не произведёт его инициализацию и запуск run().
* **Runnable** – после запуска поток переходит в это состояние. В этот момент происходит выполнение метода run(), то есть поток выполняет свою задачу.
* **Waiting** – состояние ожидания какого-либо события, например, доступа к внешнему файлу. Поток ожидает сигнала из других потоков, и до тех пор не производит никаких действий.
* **Timed Waiting** – состояние ожидания по времени, то есть поток переходит в состояние ожидания на какой-то интервал времени, после чего возвращается в состояние Runnable.
* **Terminated (Dead)** – состояние завершения потока.

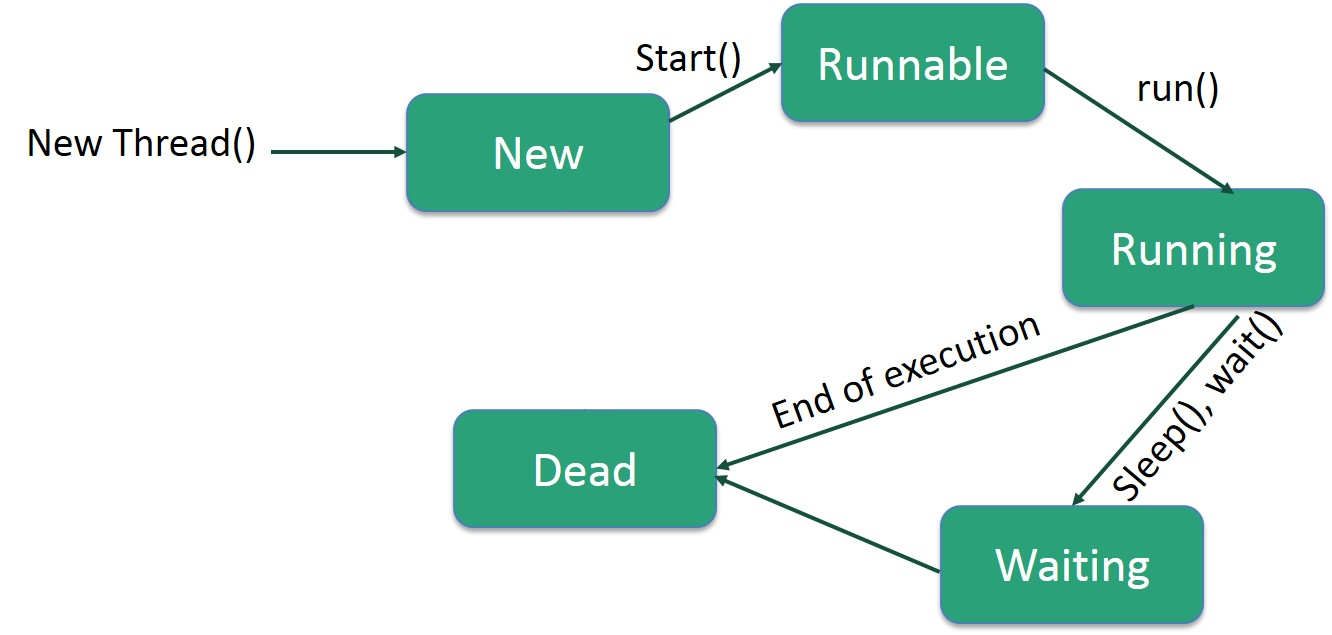


Рисунок 8 – Жизненный цикл потока в Java

* 1. Приоритеты потоков

У каждого потока Java есть приоритет, который помогает операционной системе определить порядок, в котором запланированы потоки. Приоритеты потоков Java находятся в диапазоне между MIN\_PRIORITY (константа 1) и MAX\_PRIORITY (константа 10). По умолчанию каждому потоку присваивается приоритет NORMAL\_PRIORITY (константа 5).

Потоки с более высоким приоритетом более важны для программы и должны быть выполнены раньше, чем потоки с более низким приоритетом. Однако приоритеты потоков не могут гарантировать порядок, в котором выполняются потоки, и в значительной степени зависят от операционной системы.

Методы потоков

* getName(): получение имени потока
* setName(): установка имени потока
* getPriority(): получение приоритета потока
* isAlive(): определяет, работает ли поток
* join(): текущий поток вызывает этот метод в другом потоке, блокируя текущий поток, пока другой поток не завершится или не пройдет указанное число миллисекунд
* sleep(): приостанавливает поток на указанное время
* start(): инициализация потока и запуск его метода run()
* yield(): заставляет процессор переключиться на обработку других потоков системы.

Как и в Python в Java есть особый вид потоков, которые называются демонами. Объявить поток демоном достаточно просто — нужно перед запуском потока вызвать его метод setDaemon(true). Проверить, является ли поток демоном, можно вызвав его метод isDaemon (). 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.

* 1. Синхронизация