МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

И. В. Курбатова, А. В. Печкуров

Язык программирования Java (глава "Многопоточное программирование", фрагмент)

Учебное пособие

Воронеж
Издательский дом ВГУ
2019

Оглавление

1 Многопоточное программирование **4**

1 Процессы 5
2 Потоки 7
3 Модель памяти Java 7
4 Запуск потоков 8
5 Методы получения информации о потоке 11
6 Потоки-демоны 14
7 Методы управления потоком 16
7.1 Методы sleep() и join() 18
7.2 Метод yield() 19
7.3 Методы interrupt(), interrupted() и isInterrupted() . . 21
8 Обработка исключений в дополнительных потоках 24
9 Пулы потоков 29
10 Синхронизация потоков 33
10.1 Отношение happens-before 37
10.2 Синхронизированные методы и блоки (synchronized) 38
10.3 Методы wait(), notify() и notifyAll() класса Object . 42
10.4 Ключевое слово volatile 44
11 Основные возможности библиотеки java.util.concurrent . . 47
12 Дополнительные возможности многопоточного программирования 49
13 Пример класса Семафор 52

Глава 1

Многопоточное программирование

Вместе с бурным ростом производительности, а затем и количества ядер компьютеров, произошедшим за последние несколько десятилетий, проблема оптимального использования программами доступных вычислительных ресурсов стала актуальной, как никогда. За счет оптимальной загрузки вычислительных ядер, посредством распараллеливания задач, можно сократить время сложных вычислений в несколько раз или, например, увеличить количество запросов, обрабатываемых сетевым приложением за единицу времени. Для этих целей Java предлагает возможности кросс-платформенного многопоточного программирования.

Но прежде чем перейти к основному содержанию главы, мы поговорим о параллелизме и конкурентности, двух важных понятиях, связанных с многопоточным програмированием.

Параллелизм ("parallelism") — это *свойство* систем, при котором вычисления выполняются одновременно и при этом, возможно, взаимодействуют друг с другом. Подчеркнем, что параллелизм подразумевает одновременное (параллельное) выполнение вычислений.

Конкурентность ("concurrency") — это *возможность* построения программ из неких независимых блоков (единиц вычислений), способных выполняться в произвольном или частично определенном порядке, без влияния на конечный результат выполнения. Подчеркнем, что конкурентность допускает (но не требует) пареллельное выполнение некоторых вычислений. С точки зрения программирования, конкурентность определяется некой моделью. Примерами таких моделей могут быть многопоточное программирование, модель взаимодействующих последовательных процессов ("communicating sequential processes (CSP)"), модель акторов ("actors") и другие модели.

В русскоязычной технической литературе эти термины нередко путают

друг с другом. Действительно, параллелизм и конкурентность являются смежными понятиями и подразумевают возможность одновременного выполнения вычислений. Однако, конкурентность является парадигмой проектирования программ и систем, т.е. в некотором смысле более общим, логическим понятием, поскольку конкурентные программы и системы обеспечивают параллелизм при условии поддержки параллелизма на физическом уровне, например, при наличии у процессора нескольких вычислительных ядер. При этом, возможна и конкурентность без параллелизма, например, многозадачность на одноядерном процессоре за счет разделения времени ("time-sharing") между задачами.

В данной главе мы сосредоточимся на модели многопоточного программирования и поговорим о возможностях многопоточного программирования в Java. Безусловно, написание производительных и, что еще более важно, надежных и предсказуемых многопоточных программ — это весьма сложная и обширная тема. В рамках этого курса мы ставим задачу ознакомить читателя с базовыми понятиями, сформировав "фундамент" для дальнейшего самостоятельного изучения темы. Начнем мы с базовых для многопоточных программ понятий — процесс и поток.

1 Процессы

Процесс ("process") — это экземпляр программы, выполняемый в текущий момент в рамках операционной системы (ОС). В то время как компьютерная программа это статический набор машинных инструкций, процесс подразумевает выполнение этих инструкций. Поэтому процесс содержит программный код и его состояние. Кроме того, с одной и той же программой могут быть ассоциированы несколько процессов. Например, при запуске программы несколько раз ОС стартует несколько процессов.

Благодаря операционной системе, процессы изолированы друг от друга, т.е. прямой доступ к памяти другого процесса невозможен. Это достигается за счет виртуального адресного пространства, индивидуально выделяемого каждому процессу. В момент запуска программы операционная система создает процесс, выделяет память и загружает в адресное пространство код и данные программы, затем запускает главный поток нового процесса. При необходимости ОС может изменить размер памяти, выделенной процессу.

Процессы всех программ выполняются в операционной системе в рам-

ках многозадачности, т.е. конкурентно. Все они конкурируют за ресурсы компьютера: вычислительные ядра, память, устройства ввода/вывода. В современных ОС многозадачность, как правило, реализуется посредством разделения времени между задачами, заключающегося в выделении некой квоты на выполнение очередного процесса, после исчерпания которой происходит переключение контекста и процессор или его ядро начинает выполнять следующий процесс. За счет такой конкурентной многозадачности у пользователей ОС возникает ощущение, что даже на процессоре с одним ядром процессы выполняются одновременно. Иногда такой принцип выполнения вычислительных задач называют "псевдопараллелизмом".

Каждый процесс с точки зрения операционной системы характеризуется следующими ресурсами:

- Образ машинных инструкций, соответствующий программе, иначе говоря, машинный код программы.
- Память, выделенная процессу (виртуальное адресное пространство). Содержит машинный код, данные процесса (например, стандартные потоки ввода/вывода), стек вызовов ("call stack") и кучу ("heap").
- Дескрипторы ресурсов, выделенных процессу. Примером таких ресурсов в Linux являются файловые дескрипторы, т.е. абстрактные идентификаторы, используемые процессом для доступа к файлу или иному ресурсу ввода/вывода, такому как сетевой сокет.
- Атрибуты безопасности, например, идентификатор пользователя, являющегося владельцем процесса, и набор операций, разрешенных процессу.
- Контекст процесса. Включает в себя содержимое пользовательского адресного пространства, аппаратных регистров процессора и структуры данных ядра, связанные с этим процессом.

Операционная система отслеживает ресурсы процессов, обеспечивая их изолированность и, при необходимости, выделяя дополнительные ресурсы. Кроме того, ОС также предоставляет средства межпроцессного взаимодействия ("inter-process communication" или "IPC"), позволяющие процессам обмениваться данными.

Отметим, что в большинстве операционных систем процесс может содержать в себе несколько потоков, в рамках которых независимо выполняются инструкции. Платформа Java подразумевает поддержку именно таких ОС.

2 Потоки

Поток ("thread") — это наименьшая последовательность программных инструкций, выполняемая операционной системой при помощи планировщика процессов ("process scheduler"). Любой процесс состоит из как минимум одного потока, который часто называют главным. Процесс, т.е. программа, будучи запущенным, может создавать дополнительные потоки.

В отличие от процессов, все потоки одного процесса имеют совместный доступ к ресурсам процесса, в первую очередь, к памяти, т.е. они взаимодействуют в одном адресном пространстве. Каждому потоку при его создании операционной системой выделяется в памяти отдельная область, в которой содержится стек вызовов потока. Его зачастую называют просто стеком потока ("thread stack"). В Java стек потока используется виртуальной машиной для хранения параметров, локальных переменных, результатов промежуточных вычислений и других данных.

Как и в случае с процессами, операционная система конкурентно выполняет потоки текущего процесса. Кроме того, во многих программах потоки конкурируют за получение доступа к определенным участкам памяти для чтения и/или записи, например, для получения текущего значения и последующего увеличения какого-либо счетчика. Обеспечение детерменированного и безошибочного доступа к оперативной памяти из нескольких потоков является основной сложностью многопоточного программирования. Для корректной работы многопоточной программы используют механизмы синхронизации памяти, которые мы рассмотрим в следующих разделах.

3 Модель памяти Java

Прежде чем перейти к рассмотрению возможностей многопоточного программирования, мы поговорим об основной спецификации, гарантирующей корректность и безопасность многопоточных программ в Java.

Модель памяти Java (“Java Memory Model” или “JMM”) — это спецификация, описывающая взаимодействие потоков с памятью в рамках JVM. Помимо освещения работы в рамках одного потока, модель памяти задает семантику языка Java при работе с памятью. Эта семантика описывает гарантии, на которые должен рассчитывать Java программист, и то, на что рассчитывать не стоит.

Оригинальная модель памяти Java, разработанная в 1995 году, являлась одной из первых попыток описания кросс-платформенной модели памяти и оказалась не самой удачной по ряду причин. В 2004 году, в рамках Java 5, ее существенно переработали. Мы не будем описывать все особенности данной спецификации, однако, не упомянуть ее в данной главе мы не могли. Любознательному читателю предлагаем ознакомиться с обзорной веб-страницей, отвечающей на основные вопросы про JMM: `http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html`. Кроме того, в качестве дополнительного чтения рекомендуем ознакомиться с главой ”Threads and Locks” спецификации ”Java Language Specification, Java SE 8”.

Перейдем, наконец, к рассмотрению основного синтаксиса работы с потоками и синхронизации памяти.

4 Запуск потоков

Как мы уже знаем, у каждого процесса есть как минимум один поток, называемый главным (“main thread”). Помимо автоматически создаваемого главного потока, программа может запускать дополнительные потоки. Все потоки в Java представлены классом `java.lang.Thread`, реализующим интерфейс `java.lang.Runnable`. Объекты класса `Thread` являются абстракцией или объектным представлением системных потоков на уровне операционной системы.

Сначала мы обсудим интерфейс `Runnable`. Этот интерфейс предназначен для реализации классами, которые предназначены для выполнения в отдельном потоке. Он крайне прост и выглядит следующим образом:

```
public interface Runnable {

    void run();

}
```

Запустить поток в Java можно двумя способами. Первый способ предполагает создание объекта класса `Thread` через вызов конструктора, принимающего на вход реализацию интерфейса `Runnable`. Затем у объекта `Thread` потребуется вызвать метод `start()`. После вызова метода `start()` для созданного потока, главный поток продолжает свое выполнение, т.е. оба потока продолжают работу параллельно. Приведем простой пример, демонстрирующий этот способ:

```
public class Main {

    public static void main(String[] args) {
        System.out.println("Главный поток: "
                + Thread.currentThread());
        // определение потока с Runnable
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("Поток запущен: "
                        + Thread.currentThread());
            }
        });
        // теперь запускаем поток
        thread.start();
    }

}
```

Данная программа выведет в консоль примерно следующее:

```
Главный поток: Thread[main,5,main]
Поток запущен: Thread[Thread-0,5,main]
```

В этом примере мы использовали анонимный класс для реализации `Runnable`, но, конечно, ничто не мешает определить реализацию в отдельном классе. Кроме того, данный пример демонстрирует использование статического метода `currentThread()`, доступного в классе `Thread`. Этот метод возвращает объект `Thread`, соответствующий текущему потоку выполнения, на котором он вызван. Поэтому в примере при выводе в консоль мы видим главным поток, запущенный при старте программы, и дополнительный поток, который стартовали мы сами.

Поговорим теперь о втором способе запуска потока. Он заключается в создании потомка класса `Thread` и переопределении метода `run()`. Демонстрация этого способа, аналогичная предыдущему примеру по логике выполнения, будет выглядеть так:

```
public class Main {

    public static void main(String[] args) {
        System.out.println("Главный поток: "
                + Thread.currentThread());
        // определение потока без Runnable
        Thread thread = new Thread() {
            @Override
            public void run() {
                System.out.println("Поток запущен: "
                        + Thread.currentThread());
            }
        };
        // теперь запускаем поток
        thread.start();
    }

}
```

Снова обратим внимание на использование анонимного класса, примененное из соображений лаконичности.

Заметим, что аналогичным образом потоки можно запускать не только из главного, но и из любого другого потока. Например, в случае наших примеров, мы легко могли бы запустить еще один поток из нашего дополнительного потока.

Обсудим теперь важные особенности поведения потоков. Дополнительный поток будет завершен, как только метод `run()` завершит свое выполнение. Запустить один и тот же поток (объект `Thread`) несколько раз параллельно или последовательно невозможно, поскольку у каждого потока есть определенный жизненный цикл. Java отслеживает текущее состояние потока, значения которого задаются `enum State` (о нем мы поговорим далее), и выбрасывает исключение `java.lang.IllegalThreadStateException` при попытке повторного запуска.

5 Методы получения информации о потоке

Класс `Thread` содержит ряд методов, необходимых для осуществления действий с самим потоком и взаимодействия данного потока с другими. С некоторыми методами, такими как `start()` и `currentThread()`, мы уже познакомились. В данном разделе мы рассмотрим методы, связанные с получением информации о потоке, а также заданием его поведения.

Рассмотрим некоторые соответствующие методы данного класса:

- `static Thread currentThread()` — возвращает текущий поток.
- `String getName()` — возвращает имя потока.
- `void setName(String name)` — задает имя потока. Альтернативно имя потока можно задать через один из конструкторов класса. Если имя потока не было задано, оно будет сформировано автоматически по шаблону вида `Thread-<порядковый-номер>`.
- `long getId()` — возвращает идентификатор потока. Это положительное число, генерируемое при конструировании потока. Идентификатор является уникальным в рамках потока и остается неизменным до момента завершения потока. Ранее использованные значения идентификатора, соответствующие завершенным потокам, могут быть повторно использованы для новых потоков.
- `void setPriority(int newPriority)` — задает приоритет потока. В случае, если поток относится к группе потоков[1], имеющей ограничение на максимальное значение приоритета, будет выбрано наименьшее из двух значений, `newPriority` и максимального значения для группы. В случае, если текущий поток, в рамках которого вызван метод, не имеет прав для изменения приоритета данного потока, будет выброшено исключение `java.lang.SecurityException`.
- `int getPriority()` — возвращает приоритет потока. Любой поток имеет приоритет. Потоки с более высоким приоритетом выбираются

[1]Группы потоков представлены классом `java.lang.ThreadGroup`. Каждая группа потоков представляет собой набор потоков и позволяет задавать и изменять некоторые общие характеристики для входящих в этот набор потоков. Группы могут быть связаны иерархией. Поток, находящийся в группе, может получить информацию о своей группе и влиять только на потоки своей группы, но к ее родительской группе у него уже не будет доступа. Таким образом, группы потоков используются для управления набором потоков и обеспечения безопасности доступа. Мы не будем подробно рассматривать группы потоков и предлагаем читателю сделать это самостоятельно.

планировщиком для выполнения с большим предпочтением, нежели потоки с меньшим приоритетом. По умолчанию, приоритет потока равен приоритету потока, в рамках которого он был создан.

- `StackTraceElement[] getStackTrace()` — возвращает массив, содержащий снимок стека вызовов потока. Если поток еще не запущен, то массив будет пустым. В случае, если метод вызван в потоке, отличном от данного, будут выполнены проверки безопасности доступа.

- `State getState()` — возвращает состояние потока. Данный метод предназначен для мониторинга состояния потоков, а не для их синхронизации. Состояние описывается `enum State` со следующими возможными значениями: `NEW` — поток еще не был запущен; `RUNNABLE` — поток запущен; `BLOCKED` — поток дожидается освобождения монитора[2]; `WAITING` — поток ожидает какого-то действия от другого потока (например, у потока был вызван метод `join()`); `TIMED_WAITING` — поток ожидает какого-то действия от другого потока и при этом задано ограничение по времени ожидания; `TERMINATED` — поток завершил выполнение.

Рассмотрим пример использования некоторых из этих методов для получения информации о потоке:

```
public class Main {

    public static void main(String[] args) {
        System.out.println("Главный поток");
        printInfo(Thread.currentThread());

        // определение доп-го потока
        Thread thread = new Thread() {
            @Override
            public void run() {
                System.out.println("Доп-й поток: внутри");
                printInfo(Thread.currentThread());
            }
        };
```

[2]О синхронизации потоков и, в частности, о блокировках и мониторах мы поговорим в следующих разделах.

```
        System.out.println("Доп-й поток: до");
        printInfo(thread);

        // изменяем и стартуем поток
        thread.setName("MyThread");
        thread.setPriority(Thread.MAX_PRIORITY);
        thread.start();

        try {
            // дожидаемся завершения работы доп-го потока
            thread.join();
            System.out.println("Доп-й поток: после");
            printInfo(thread);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private static void printInfo(Thread thread) {
        System.out.printf(
                "Имя: %s, id: %d, " +
                        "приоритет: %d, состояние: %s\n",
                thread.getName(),
                thread.getId(),
                thread.getPriority(),
                thread.getState()
        );
    }

}
```

После запуска данная программа выведет в консоль следующее:

```
Главный поток
Имя: main, id: 1, приоритет: 5, состояние: RUNNABLE
Доп-й поток: до
Имя: Thread-0, id: 11, приоритет: 5, состояние: NEW
Доп-й поток: внутри
Имя: MyThread, id: 11, приоритет: 10, состояние: RUNNABLE
```

```
Доп-й поток: после
Имя: MyThread, id: 11, приоритет: 10, состояние: TERMINATED
```

Этот пример достаточно прямолинеен и наглядно демонстрирует получение базовой информации о потоке, а также изменение имени и приоритета потока. Отметим лишь использование метода `join()` для целей демонстрации статуса завершенного потока. Мы обсудим этот метод в следующих разделах.

6 Потоки-демоны

Как мы уже знаем, каждый поток имеет определенное значение приоритета, который системный планировщик учитывает при выборе потоков для исполнения. Кроме того, поток может быть отмечен как поток-демон ("daemon thread"). Этот признак учитывается при определении условий завершения работы JVM. Виртуальная машина продолжает выполнять потоки программы до наступления одного из следующих условий:

- Один из потоков вызвал метод `exit()` класса `Runtime`. При этом менеджер безопасности ("security manager"), специальный класс, задающий политики безопасности, подтвердил права на вызов.
- Все потоки, не являющиеся демонами, завершили выполнение.

Таким образом, потоки, отмеченные как демоны не препятствуют завершению Java-программы. Эти потоки могут быть удобны для выполнения побочных, необязательных задач в отдельном потоке. Примером таких задач может быть отправка записей журнала приложения по сети или сбор и обработка информации, необходимой для мониторинга состояния приложения.

Ниже перечислены методы класса `Thread`, связанные с потоками-демонами:

- `boolean isDaemon()` — возвращает признак того, что текущий поток является потоком-демоном. По умолчанию значение признака равно значению для потока, в рамках которого он был создан.
- `void setDaemon(boolean on)` — переопределяет значение признака потока-демона. Должен быть вызван до запуска потока. В противном случае выбрасывает исключение `java.lang.IllegalThreadStateException`.

Еще раз отметим, что при создании новый поток по умолчанию получает такое же значение приоритета и признака потока-демона, как у текущего потока.

Рассмотрим пример использования потоков-демонов:

```
public class Main {

    public static void main(String[] args) {
        Thread thread = new Thread() {
            @Override
            public void run() {
                while (true) {
                    System.out.println("Доп-й поток: " +
                            "проводим вычисления");
                    // некие вычисления
                    int fib100 = fib(5);
                    System.out.printf("Доп-й поток: результат " +
                            "вычислений: %d\n", fib100);
                }
            }
        };
        System.out.println("Стартуем доп-й поток-демон");
        thread.setDaemon(true);
        thread.start();

        // некие вычисления
        // (чуть более длительные, чем в доп-м потоке)
        fib(15);

        System.out.println("Программа завершилась");
    }

    // вычисление чисел Фибоначчи через рекурсию
    // (имеет экспоненциальное время выполнения)
    private static int fib(int num) {
        if (num == 0) return 0;
        if (num == 1) return 1;
        return fib(num - 2) + fib(num - 1);
```

```
    }

}
```

Этот пример выведет в консоль примерно следующее (результат зависит от многих факторов и может варьироваться):

```
Стартуем доп-й поток-демон
Доп-й поток: проводим вычисления
Доп-й поток: результат вычислений: 5
Доп-й поток: проводим вычисления
Программа завершилась
Доп-й поток: результат вычислений: 5
Доп-й поток: проводим вычисления
Доп-й поток: результат вычислений: 5
Доп-й поток: проводим вычисления
```

В нашем примере мы создали дополнительный поток-демон, в котором происходят некие условно бесконечные вычисления. Пример демонстрирует то, что выполнение потока-демона не препятствует завершению программы в тот момент, когда все обычные потоки (в нашем примере это только главный поток) завершают выполнение.

7 Методы управления потоком

Теперь мы рассмотрим способы управления потоком. Эти методы позволяют запускать, временно приостанавливать и, при некоторых условиях, даже полностью останавливать выполнение потока. Большинство из этих методов, например, метод `sleep()`, рассчитаны на вызов внутри кода потока, т.е. в теле метода `run()`. Некоторые методы, такие как `start()` или `interrupt()`, рассчитаны на вызов в рамках другого потока.

Перечислим основные методы, позволяющие управлять потоком:

- `void start()` — запускает выполнение потока. В случае попытки повторного запуска потока будет выброшено исключение `java.lang.IllegalThrea`
- `static void sleep(long millis)` — приостанавливает выполнение потока на указанное время (единица изменения - миллисекунды). При этом начинают выполняться другие потоки или процессы. Этот метод и некоторые дальнейшие выбрасывает исключение `java.lang.Interrupte`

в случае, если поток был прерван (“interrupted”). Точность соответствия желаемого и фактического периода бездействия потока зависит от точности системного планировщика и механизма определения времени. О прерывании работы потоков мы поговорим далее.

- `void join(long millis)` — приводит к ожиданию завершения выполнения потока в течение указанного времени. Время, указанное в миллисекундах, обозначает максимальное время ожидания. Если выполнение потока не завершится в течение обозначенного интервала, то текущий поток продолжит выполнение. Этот метод полезен для случаев, когда в одном потоке требуется дождаться завершения работы другого потока. Выбрасывает исключение `InterruptedException` в случае, если поток был прерван (“interrupted”).

- `void join()` — аналогичен предыдущего методу, однако, приводит к неограниченному по времени ожиданию.

- `static void yield()` — дает системному планировщику подсказку о том, что текущий поток завершил значимую часть работы и можно переключиться на выполнение какого-либо другого потока или процесса. Планировщик может проигнорировать эту подсказку и продолжить выполнение текущего потока. Данный метод будет полезен в тех случаях, когда требуется дать возможность выполниться другому потоку. Примером такой ситуации может быть выполнение вычислительно сложной задачи. Вычисления при этом можно разбить на части и вызывать метод `yield()` перед продолжением вычислений. Это позволит другим потокам или процессам выполняться периодически. Заметим, что это узкоспециализированный метод, поэтому его использование должно быть обоснованным и должно сопровождаться соответствующим тестированием и профилированием программы.

- `void interrupt()` — отправляет в поток ”сигнал” прерывания. При этом, в случае, если прерываемый поток был приостановлен (заблокирован) вызовом таких методов, как `sleep()`, `join()` или вариации метода `wait()`, его выполнение будет прервано выбросом исключения `InterruptedException`. В случае попытки неразрешенного доступа к потоку будет выброшено исключение `java.lang.SecurityException`.

- `static boolean interrupted()` — возвращает признак того, что текущий поток был прерван. В отличие от вызова метода `isInterrupted()`

в результате вызова данного метода в объекте потока признак сбрасывается на значение `false`.

- `boolean isInterrupted()` — возвращает признак того, что текущий поток был прерван. Значение признака при этом не сбрасывается.

Эти и другие, рассмотренные ранее методы класса `Thread` — это далеко не полный список методов, доступных в данном классе, но даже такой, неполный список сложно запомнить и понять принцип работы только по описанию. Поэтому мы последовательно рассмотрим использование методов управления потоком на примерах.

7.1 Методы sleep() и join()

Для начала рассмотрим пример использования методов `sleep()` и `join()`. Метод `sleep()` приостанавливает выполнение потока, на заданное время. Метод `join()` позволяет приостановить выполнение текущего потока до момента завершения выполнения заданного потока.

В нашем примере основной поток будет запускать дополнительный поток, в котором мы будем приостанавливать выполнение через вызов `sleep()`. После чего основной поток будет дожидаться завершения дополнительного потока при помощи вызова `join()`. Эта программа выглядит следующим образом:

```
public class Main {

    public static void main(String[] args) {
        Thread thread = new Thread() {
            @Override
            public void run() {
                System.out.println("Доп-й поток: стартовал");
                // ждем 1 секунду
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Доп-й поток: завершился");
            }
```

```
        };
        System.out.println("Стартуем доп-й поток");
        thread.start();

        try {
            // дожидаемся завершения работы доп-го потока
            thread.join();
            System.out.println("Программа завершилась");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

Эта программа выведет в консоль следующее:

```
Стартуем доп-й поток
Доп-й поток: стартовал
Доп-й поток: завершился
Программа завершилась
```

7.2 Метод yield()

Рассмотрим теперь метод `yield()`. Как мы уже отмечали, это узкоспециализированный метод, который дает системному планировщику подсказку о том, что текущий поток завершил значимую часть работы и можно переключиться на выполнение какого-либо другого потока или процесса. Поэтому продемонстрировать его работу будет относительно сложно.

Мы рассмотрим программу, эмулирующий вычислительно сложную задачу, выполняемую в дополнительном потоке. При этом дополнительный поток будет периодически вызывать метод `yield()` для того, чтобы планировщик мог передать управление другому потоку или процессу. Заметим, что заметные отличия в работе этой программы без вызова `yield()` и с ним с большей вероятностью удастся увидеть на компьютерах, где программе доступно только одно ядро процессора.

Приведем ниже код нашего примера:

```
public class Main {

    public static void main(String[] args) {
        Thread thread = new Thread() {
            @Override
            public void run() {
                System.out.println("Доп-й поток: стартовал");
                // делаем 10 итераций с вычислением
                // 40го числа Фибоначчи
                heavyCalc(10, 40);
                System.out.println("Доп-й поток: завершился");
            }
        };
        System.out.println("Стартуем доп-й поток");
        thread.start();

        // делаем такое же вычисление, что и в доп-м потоке
        heavyCalc(10, 40);
        System.out.println("Осн-й поток: завершился");
    }

    // вычисление чисел Фибоначчи через рекурсию
    // (имеет экспоненциальное время выполнения)
    private static int fib(int num) {
        if (num == 0) return 0;
        if (num == 1) return 1;
        return fib(num - 2) + fib(num - 1);
    }

    private static void heavyCalc(int iterations, int fibNum) {
        for (int it = 0; it < iterations; it++) {
            long time = System.currentTimeMillis();
            fib(fibNum);
            System.out.printf("Поток %s: вычисление " +
                              "заняло %d мс\n",
                    Thread.currentThread(),
                    (System.currentTimeMillis() - time));
```

```
            // предлагаем передать выполнение
            // другому потоку/процессу
            Thread.yield();
        }
    }

}
```

Эта программа выведет в консоль следующее (вывод немного сокращен):

```
Стартуем доп-й поток
Доп-й поток: стартовал
Поток Thread[Thread-0,5,main]: вычисление заняло 1037 мс
Поток Thread[main,5,main]: вычисление заняло 1063 мс
Поток Thread[Thread-0,5,main]: вычисление заняло 944 мс
Поток Thread[main,5,main]: вычисление заняло 945 мс
...
Доп-й поток: завершился
Поток Thread[main,5,main]: вычисление заняло 851 мс
Осн-й поток: завершился
```

В консольном выводе мы видим периодическое переключение между потоками. Подчеркнем еще раз, что `yield()` на практике используется редко и только при осмысленной и обоснованной необходимости. Однако, не обсудить этот метод и его предназначение мы не могли.

7.3 Методы interrupt(), interrupted() и isInterrupted()

Теперь мы рассмотрим методы `interrupt()`, `interrupted()` и `isInterrupted()`. Эти методы предназначены для прерывания работы запущенного потока. При этом ничто не мешает реализовать поток так, чтобы он игнорировал полученный сигнал о прерывании и продолжал работу[3]. Другими словами, для прерывания работы потока требуются согласованные действия со стороны обоих потоков, прерывающего и прерываемого. Такой подход к прерыванию работы потоков называют кооперативным. Помимо кооперативного способа, в Java нет других способов принудительной остановки

[3]Такая реализация считается плохим тоном. Потоки должны корректно обрабатывать сигналы прерывания работы.

потока[4].

Напомним, что метод `interrupt()` отправляет в поток сигнал прерывания, что приводит к присвоения значения `true` признака того, что текущий поток был прерван, и к возможному выбросу исключения `InterruptedExceptio` из ряда методов класса `Thread`. Методы `interrupted()` и `isInterrupted()` возвращают значение признака того, что текущий поток был прерван. При этом вызов метода `interrupted()` сбрасывает значение признака на `false`.

Для демонстрации этих методов мы рассмотрим пример, где основной поток будет запускать два дополнительных потока и пытаться завершить их работу. Код программы приведен ниже:

```
public class LooperThread extends Thread {

    @Override
    public void run() {
        while (true) {
            // проверка сигнала прерывания
            if (interrupted()) {
                System.out.println("Доп-й поток " +
                        "LooperThread: получен сигнал");
                break;
            }
            // здесь могли бы быть какие-то вычисления
        }
        System.out.println("Доп-й поток " +
                "LooperThread: завершился");
    }

}

public class SleeperThread extends Thread {

    @Override
    public void run() {
```

[4]В первых версиях Java в классе `Thread` был метод `stop()`, принудительно останавливающий поток. Однако, применение этого метода часто приводило к неопределенному состоянию данных в памяти работающего приложения. Поэтому позднее метод `stop()` объявили устаревшим и добавили средства кооперативного прерывания работы потока.

```
        while (true) {
            try {
                // ждем 10 сек
                Thread.sleep(10_000);
            } catch (InterruptedException e) {
                System.out.println("Доп-й поток " +
                        "SleeperThread: " +
                        "получен сигнал-исключение");
                break;
            }
            // проверка сигнала нужна
            if (interrupted()) {
                System.out.println("Доп-й поток " +
                        "SleeperThread: получен сигнал");
                break;
            }
        }
        System.out.println("Доп-й поток " +
                "SleeperThread: завершился");
    }

}

public class Main {

    public static void main(String[] args)
            throws InterruptedException {
        Thread lt = new LooperThread();
        lt.start();
        // немного ждем и пытаемся остановить поток
        Thread.sleep(100);
        lt.interrupt();
        lt.join();
        System.out.println("Осн-й поток: дождался " +
                "остановки первого доп-го потока");

        Thread st = new SleeperThread();
```

```
        st.start();
        // немного ждем и пытаемся остановить поток
        Thread.sleep(100);
        st.interrupt();
        st.join();
        System.out.println("Осн-й поток: дождался " +
                "остановки второго доп-го потока");

        System.out.println("Осн-й поток: завершился");
    }

}
```

Эта программа выведет в консоль следующее:

```
Доп-й поток LooperThread: получен сигнал
Доп-й поток LooperThread: завершился
Осн-й поток: дождался остановки первого доп-го потока
Доп-й поток SleeperThread: получен сигнал-исключение
Доп-й поток SleeperThread: завершился
Осн-й поток: дождался остановки второго доп-го потока
Осн-й поток: завершился
```

Отметим, что в методе `run()` класса `SleeperThread` обрабатывается как исключение `InterruptedException`, так и признак (или сигнал) прерывания работы потока, возвращаемый методом `interrupted()`. Одной обработки исключения недостаточно, поскольку нет никаких гарантий, что выполнение потока будет находиться внутри метода `sleep()` или аналогичного метода, выбрасывающего `InterruptedException`, во время получения сигнала прерывания. Кроме того, при выбросе исключения `InterruptedException` признак прерывания, возвращаемый методами `interrupted()` и `isInterrupted()`, будет иметь значение `false`, поскольку в этом случае таким признаком является само выброшенное исключение.

8 Обработка исключений в дополнительных потоках

Обработка исключений в дополнительных потоках имеет свои особенности в сравнении с обработкой исключений в главном потоке. Ключевая

особенность состоит в том, что в дополнительных потоках выброс необработанных исключений, относящихся к непроверяемым исключениям, не приводит к завершению работы программы, в то время как в случае главного потока работа программы завершается.

Следующий пример демонстрирует эту особенность:

```
public class Main {

    public static void main(String[] args)
            throws InterruptedException {
        Thread thread = new Thread() {
            @Override
            public void run() {
                System.out.println("Доп-й поток: " +
                        "старт");
                throw new RuntimeException();
            }
        };
        System.out.println("Стартуем доп-й поток");
        thread.start();

        // ждем 3 секунды до момента завершения программы
        Thread.sleep(3000);
        System.out.println("Программа завершилась");
    }

}
```

Эта программа выведет в консоль следующее:

```
Стартуем доп-й поток
Доп-й поток: старт
Exception in thread "Thread-0" java.lang.RuntimeException
at Main$1.run(Main.java:10)
Программа завершилась
```

Из консольного вывода мы видим, что выброс необработанного исключения `RuntimeException` в дополнительном потоке привел только к записи информации (причем эта запись ушла в стандартный поток для ошибок). Программа при этом продолжила выполнение.

Конечно, для более контролируемого выполнения программы можно использовать блок `try/catch` на верхнем уровне метода `run()` потока. Однако, в Java есть и более удобные способы описания обработчиков исключений для дополнительных потоков. Перечислим соответствующие методы класса `Thread`:

- `void setUncaughtExceptionHandler(UncaughtExceptionHandler eh)` — задает обработчик исключений, вызываемый при выбросе необработанных исключений в данном потоке. В случае если обработчик для потока не задан, используется обработчик группы потоков (`ThreadGroup`), к которой относится данный поток (при ее наличии).
- `UncaughtExceptionHandler getUncaughtExceptionHandler()` — возвращает обработчик, заданный для данного потока. В случае отсутствия обработчика возвращается обработчик группы потоков.
- `static void setDefaultUncaughtExceptionHandler(UncaughtExceptionHandl` задает обработчик исключений по умолчанию. В случае выброса необработанного исключения сначала проверяется наличие обработчика уровня потока, затем — уровня группы потоков, после чего обработчика по умолчанию. Только первый найденный в этой цепочке обработчик будет вызван.

Эти методы используют интерфейс `UncaughtExceptionHandler`, являющийся вложенным интерфейсом класса `Thread`. Он содержит один метод и выглядит следующим образом:

```
public interface UncaughtExceptionHandler {

    void uncaughtException(Thread t, Throwable e);

}
```

Метод `uncaughtException()` обработчика будет вызван перед завершением работы потока, в котором было выброшено необработанное исключение. При этом любые необработанные исключения, выброшенные в ходе выполнения самого метода `uncaughtException()` будут проигнорированы. Отметим также, что класс `ThreadGroup` реализует интерфейс `UncaughtExceptionHandler` и, таким образом, сам по себе является обработчиком исключений.

Рассмотрим теперь пример использования разных видов обработчиков исключений:

```
public class Main {

    public static void main(String[] args)
            throws InterruptedException {
        Thread.setDefaultUncaughtExceptionHandler(
                new Handler("По умолчанию"));

        Thread t1 = createThread(1);
        t1.setUncaughtExceptionHandler(
                new Handler("Обычный"));
        System.out.printf("Стартуем доп-й поток 1: %s\n", t1);
        t1.start();

        Thread t2 = createThread(2);
        System.out.printf("Стартуем доп-й поток 2: %s\n", t2);
        t2.start();

        // ждем завершения обоих потоков
        t1.join();
        t2.join();
        System.out.println("Программа завершилась");
    }

    private static Thread createThread(int num) {
        return new Thread() {
            @Override
            public void run() {
                System.out.printf("Доп-й поток %d: " +
                        "старт\n", num);
                throw new RuntimeException();
            }
        };
    }

}
```

```
public class Handler
        implements Thread.UncaughtExceptionHandler {

    private final String name;

    public Handler(String name) {
        this.name = name;
    }

    @Override
    public void uncaughtException(Thread t,
                                  Throwable e) {
        System.out.printf("[%s] В потоке %s\n  " +
                "выброшено исключение %s\n", name, t, e);
    }

}
```

Этот пример наглядно показывает использование обработчиков исключений в дополнительных потоках. В результате работы он выведет в консоль следующее:

```
Стартуем доп-й поток 1: Thread[Thread-0,5,main]
Стартуем доп-й поток 2: Thread[Thread-1,5,main]
Доп-й поток 1: старт
Доп-й поток 2: старт
[Обычный] В потоке Thread[Thread-0,5,main]
  выброшено исключение java.lang.RuntimeException
[По умолчанию] В потоке Thread[Thread-1,5,main]
  выброшено исключение java.lang.RuntimeException
Программа завершилась
```

Исключение из потока `t2`, где не было без заданного обработчика исключений, было перехвачено обработчиком по умолчанию, тогда как исключение из потока `t1` попало в его собственный, отдельный обработчик.

9 Пулы потоков

Пулы потоков (“thread pool”) — это объекты (и классы), предоставляющие абстракцию для определенного набора потоков и позволяющие отправлять вычислительные задачи на выполнение в одном из потоков (“worker thread”), входящих в пул. В случае, если все потоки пула оказываются заняты при отправлении очередной задачи на выполнение, эта задача добавляется во внутреннюю очередь пула и выполнится позднее, когда освободится один из потоков. Размер пула, в зависимости от конкретного класса и его конфигурации, может быть как фиксированным, так и изменяющимся динамически.

Пулы потоков имеют ряд преимуществ в сравнении с ”ручным” порождением и запуском потоков, выполняющих какую-то задачу. Во-первых, обычно пул потоков реализован так, чтобы он переиспользовал потоки, а не порождал новые перед выполнением каждой задачи. Это позволяет не тратить процессорное время и память на выделение ресурсов для потока. Пулы также позволяют отложить создание потоков до появления задач и указать время, после истечения которого неиспользуемые потоки будут уничтожены, при условии отсутствия новых задач. Во-вторых, у пула потоков можно задать максимально возможное количество потоков. Это позволяет ограничить максимальный объем ресурсов, потребляемых программой. Например, для вычислительно сложных задач часто используют пул фиксированного размера, совпадающего с количеством ядер процессора. При написании программ всегда стоит продумывать поведение программы при росте размера входных данных и ограничивать максимальное потребляемое количество ресурсов, будь то количество потоков или потребляемая память.

В этом разделе мы в первую очередь хотим познакомить читателя с понятием ”пул потоков”, поэтому мы не будем рассматривать все возможности стандартной библиотеки Java, связанные с пулами потоков, и ограничимся малой их частью, необходимой для демонстрационных целей.

Рассмотрим использование пула потоков на примере класса `java.util.concurren`
Этот класс позволяет гибко конфигурировать размер пула, задавать правила создания и уничтожения потоков, входящих в пул, а также задавать очередь, в которую будут попадать задачи в ситуации, когда все потоки пула заняты. В качестве задач в этом пуле выступают экземпляры уже знакомого нам интерфейса `Runnable`. Код примера представлен ниже:

```
public class Main {

    public static void main(String[] args)
            throws InterruptedException {
        // определяем кол-во ядер процессора
        int cores = Runtime.getRuntime().availableProcessors();
        // создаем очередь с максимальным размером 100 задач
        BlockingQueue<Runnable> queue =
                new LinkedBlockingQueue<>(100);
        // создаем пул
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                // мин и макс размеры пула:
                cores, cores * 2,
                // время ожидания до уничтожения
                // неиспользуемых потоков
                60, TimeUnit.SECONDS,
                // указываем очередь
                queue,
                // указываем объект, который будет вызван,
                // если превышен размер очереди
                new RejectedHandler()
        );

        // по умолчанию пул будет создавать потоки
        // в "ленивом" режиме, но таким образом мы могли
        // бы "прогреть" пул до мин кол-ва потоков:
        // executor.prestartAllCoreThreads();

        System.out.printf("Начальный размер пула %d потоков\n",
                executor.getPoolSize());

        // добавляем в пул задачи
        int tasks = 100 + cores * 4;
        for (int i = 0; i < tasks; i++) {
            executor.execute(new FibRunnable(i));
        }
```

```
        // переключаем пул в режим завершения работы,
        // после чего новые задачи добавлять в него нельзя
        executor.shutdown();
        // дожидаемся завершения выполнения всех задач
        while (!executor.awaitTermination(10,
                TimeUnit.SECONDS)) {
            System.out.println("Ожидаем завершения");
        }
        System.out.println("Все задачи завершены");
    }

}

public class FibRunnable implements Runnable {

    private final int indx;

    public FibRunnable(int indx) {
        this.indx = indx;
    }

    @Override
    public void run() {
        System.out.printf("Стартуем задачу %d " +
                "в потоке %s\n", indx, Thread.currentThread());
        // вычисляем 40е число Фибоначчи
        fib(40);
        System.out.printf("Задача %d в потоке %s завершилась\n",
                indx, Thread.currentThread());
    }

    // вычисление чисел Фибоначчи через рекурсию
    // (имеет экспоненциальное время выполнения)
    private int fib(int num) {
        if (num == 0) return 0;
        if (num == 1) return 1;
        return fib(num - 2) + fib(num - 1);
```

```
    }

    @Override
    public String toString() {
        return "Задача " + indx;
    }

}

public class RejectedHandler
        implements RejectedExecutionHandler {

    @Override
    public void rejectedExecution(Runnable r,
                                  ThreadPoolExecutor e) {
        System.out.printf("Задача \"%s\" отклонена\n", r);
    }
}
```

После запуска эта программа выведет в консоль примерно следующее (содержимое сокращено в угоду читаемости):

```
Начальный размер пула 0 потоков
Стартуем задачу 1 в потоке Thread[pool-1-thread-2,5,main]
Стартуем задачу 4 в потоке Thread[pool-1-thread-5,5,main]
Стартуем задачу 0 в потоке Thread[pool-1-thread-1,5,main]
...
Стартуем задачу 109 в потоке Thread[pool-1-thread-10,5,main]
Стартуем задачу 2 в потоке Thread[pool-1-thread-3,5,main]
Задача "Задача 112" отклонена
...
Задача "Задача 123" отклонена
Стартуем задачу 111 в потоке Thread[pool-1-thread-12,5,main]
...
Задача 80 в потоке Thread[pool-1-thread-8,5,main] завершилась
Стартуем задачу 104 в потоке Thread[pool-1-thread-8,5,main]
Ожидаем завершения
...
Задача 105 в потоке Thread[pool-1-thread-7,5,main] завершилась
```

```
Задача 104 в потоке Thread[pool-1-thread-3,5,main] завершилась
Все задачи завершены
```

Обратите внимание на тот факт, что последовательность запуска и завершения задач в пуле может не совпадать с порядком добавления задач (в примере задача №2 начинает выполняться после задачи №109). Это обусловлено заданным поведением пула. В нашем случае при добавлении новых задач пул создает дополнительные потоки в случае, если начальному количеству потоков, равному в этом примере количеству ядер процессора, уже присвоены задачи.

Пулы потоков — это очень мощный инструмент многопоточного программирования. Они прекрасно подходят для задач, когда требуется распараллелить вычисления и/или обработку данных, при этом ограничив максимально возможное потребление ресурсов компьютера. С другой стороны, в случаях, когда вычислительные задачи зависят от данных друг друга в процессе выполнения, т.е. требуют синхронной обработки, пулы потоков могут оказаться не лучшим выбором, поскольку при неконтролируемом порядке выполнения задач может возникнуть риск блокировки.

10 Синхронизация потоков

Ранее мы рассматривали примеры многопоточных программ, где не требовалось одновременно читать и модифицировать одни и те же данные, находящиеся в памяти, из разных потоков. Однако, как только в программе появляется такая необходимость, оказывается, что для ее корректного поведения требуется использовать средства синхронизации, предоставляемые Java. Без использования средств синхронизации код, работающий корректно в рамках однопоточной программы, будет приводить к непредсказуемым результатам при использовании нескольких потоков, одновременно читающих и изменяющих одни и те же значения.

Практически все, что мы рассмотрим в дальнейших разделах, относится к так называемой пессимистичной или блокирующей синхронизации[5]. Именно ее в большинстве случаев подразумевают, когда говорят о синхронизации. Суть такой синхронизации заключается в том, что когда поток пытается получить управление каким-либо ресурсом, например, по-

[5]Отметим, что ключевое слово `volatile` не обеспечивает синхронизацию потоков, но об этом мы поговорим позже.

лучить монитор, связанный с объектом для входа в блок `synchronized`[6], поток блокируется, т.е. приостанавливает свое выполнение, до момента получения контроля над запрошенным ресурсом. Только один поток может захватить ресурс и выполнить определенную часть кода. Другими словами, использование пессимистичной синхронизации можно охарактеризовать фразой "все или ничего".

На уровне ОС инструменты синхронизации Java обеспечиваются известными примитивами такими, как мьютексы, семафоры и критические секции. Поэтому читатели, знакомые с этими примитивами, должны быстрее освоить дальнейший материал.

Поскольку блокирующая синхронизация имеет определенные накладные расходы, то логично, что традиционная проблема высокопроизводительных многопоточных программ — это сведение синхронизации до минимально необходимого уровня, позволяющего оптимально загрузить все доступные ресурсы компьютера.

Отметим также, что существует также понятие оптимистичной или неблокирующей сихронизации. Под ней подразумевают синхронизацию, основанную на попытке изменения общего ресурса, т.е. значения, находящегося в памяти. Поток при этом не блокируется и в случае, если изменение значения не удалось, поток продолжает выполнение и имеет возможность осуществить повторную попытку изменения ресурса или же выполнить какие-то другие действия. Стандартная библиотека Java предоставляет некоторые средства для неблокирующей синхронизации, например, набор классов `Atomic*` из пакета `java.util.concurrent`. Но, поскольку оптимистичная синхронизация уже выходит за рамки данной главы, мы предлагаем читателю самостоятельно ознакомиться этой темой.

В качестве демонстрации необходимости наличия синхронизации мы рассмотрим простой пример с целочисленным счетчиком, значение которого изменяется из нескольких потоков:

```
public class Main {

    private static final int ITERATIONS = 10_000;
    private static int cnt;

    public static void main(String[] args)
```

[6]Блоки и методы, отмеченные ключевым словом `synchronized`, и их особенности мы обсудим немного позже.

```
            throws InterruptedException {
        for (int i = 0; i < ITERATIONS; i++) {
            cnt = 0;
            inc();
            if (cnt != 200) {
                System.out.printf("Ошибка. " +
                        "Счетчик оказался равен %d\n", cnt);
            }
        }

        System.out.println("Программа завершена");
    }

    public static void inc()
            throws InterruptedException {
        // оба потока увеличивают счетчик на 100
        Runnable incrementer = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    cnt = cnt + 1;
                }
            }
        };
        Thread t1 = new Thread(incrementer);
        Thread t2 = new Thread(incrementer);
        // стартуем оба потока
        t1.start();
        t2.start();
        // дожидаемся их завершения
        t1.join();
        t2.join();
    }

}
```

Эта программа сбрасывает значение счетчика до ноля, а затем увеличивает его на 100 параллельно из двух потоков. Этот процесс выполняется

много раз и в случае, если итоговое значение счетчика оказывается отличным от ожидаемого значения 200, программа выводит предупреждение в консоль.

На первый взгляд эта программа выглядит безопасно, по крайней мере с точки зрения действия каждого потока в отдельности. Однако, если запустить эту программу, вы с высокой долей вероятности увидите одно или несколько предупреждений в ее консольном выводе. При этом, есть и шанс, что программа завершится без предупреждений, т.е. для всех итераций значения счетчика будут совпадать. И это лишний раз подчеркивает, насколько неожиданным и проблемным может быть воспроизведение ошибок в многопоточных программах.

Конкретная проблема в данной программе заключается в том, как именно происходит увеличение счетчика. Дело в том, что в каждом из потоков счетчик увеличивается в два шага. На первом шаге считывается текущее значение счетчика `cnt`. На втором — счетчику присваивается только что считанное значение, увеличенное на 1. Сама проблема в том, что с некоторой вероятностью один из потоков может считать значение счетчика сразу после того, как другой поток считал то же самое значение. В результате, оба потока запишу в поле `cnt` одно и то же значение, что и приводит к некорректным итоговым значениям. В рамках дальнейших разделов мы рассмотрим то, как эту программу можно превратить в корректную с точки зрения многопоточности.

Приведем пример проблемной последовательности действий:

1. Поток `t1` читает значение счетчика `cnt` из памяти. Значение `cnt` равно 11.
2. Поток `t2` читает значение счетчика `cnt` из памяти. Значение `cnt` равно 11.
3. Поток `t1` вычисляет увеличенное значение счетчика, равное 12. Значение `cnt` равно 11.
4. Поток `t2` вычисляет увеличенное значение счетчика, равное 12. Значение `cnt` равно 11.
5. Поток `t1` записывает увеличенное значение счетчика в память. Значение `cnt` равно 12.
6. Поток `t2` записывает увеличенное значение счетчика в память. Значение `cnt` равно 12.

Отметим также, что этот пример показывает одну из наиболее тривиальных проблем из класса всех подобных проблем многопоточного кода. При более сложном, и при этом также небезопасном с точки зрения многопоточности, коде проблемы могут оказаться гораздо менее наглядными и ожидаемыми.

В общем, наиболее частые проблемы, встречающиеся в некорректных с точки зрения многопоточности программах, сводятся к двум категориям. Первая категория — это состояние гонки (“race condition”), т.е. ситуация, когда данные приходят в некорректное состояние в результате несинхронизированных изменений. Рассмотренный нами пример иллюстрирует как раз эту категорию. Вторая категория — это проблемы в программах, где потоки синхронизированы, но во время взаимодействия потоков возникают ситуации, препятствующие продолжению корректной работы программы. Наиболее часто встречающаяся проблема из этой категории — это взаимная блокировка (“deadlock”), т.е. ситуация, когда два или более потока оказываются взаимно заблокированы в ожидании освобождения ресурсов, принадлежащих друг другу. В этом случае программа или некоторая ее часть прекращает свое выполнение. Существуют и другие проблемы из этой категории, но они встречаются не настолько часто, поэтому мы не будем их освещать.

Далее мы введем одно важное определение и рассмотрим основные средства синхронизации, предоставляемые Java.

10.1 Отношение happens-before

Прежде чем приступить к рассмотрению средств синхронизации, нам нужно ввести одно, важное для многопоточного программирования на языке Java понятие. Речь о так называемом отношении “happens-before” (“происходит-до”), описываемом в ”Java Memory Model” и спецификации ”Java Language Specification” (JLS) и упоминаемом в стандартной документации JDK. Это отношение между некоторыми действиями, выполняемыми в разных потоках программы. При наличии отношения “happens-before” между некоторыми действиями А и Б, все операции и данные, производимые и изменяемые действием А, будут гарантированно выполнены и видны до начала выполнения действия Б.

Поясним это на нашем предыдущем примере с целочисленным счетчиком. Напомним, что в нашей программе было два потока, выполняющихся одновременно увеличение счетчика. Каждый из потоков читал

счетчик и записывал в него увеличенное на 1 значение. При наличии отношения “happens-before” между этими действиями потоков счетчик будет корректно увеличиваться и его конечное значение всегда будет равно 200. Без этого отношения операция чтения в одном из потоков может произойти между операциями чтения и записи в другом потоке, что мы и наблюдали в примере без корректной синхронизации.

Синхронизация потоков всегда подразумевает отношение “happens-before”[7] и обеспечивается за счет того или иного набора низкоуровневых механизмов: запрета переупорядочивания инструкций JIT-компилятором, инструкций для барьеров памяти (“memory barrier” или “memory fence”), механизмов синхронизации уровня операционной системы (мьютексы, семафоры и критические секции) или атомарных инструкций для той или иной архитектуры процессора (например, CAS-инструкций[8] для процессоров x86, Itanium и Sparc). Конечно, в рамках этой главы мы не будем углубляться в эти детали. Однако, поскольку само понятие носит несколько абстрактный характер, мы будем рассматривать отношение “happens-before” в свете конкретных (высокоуровневых) механизмов синхронизации, предоставляемых Java. Мы начнем с наиболее известного из них — блоков `synchronized`.

Все инструменты языка Java, которые мы будем рассматривать в рамках этого раздела, обеспечивают отношение “happens-before”. Для этих инструментов мы поясним, к чему именно относится отношение в конкретном случае. Подчеркнем, что отношение “happens-before” в Java возникает и в некоторых других случаях, но, поскольку эта глава — обзорное введение в многопоточное программирование, мы не будем вдаваться во все детали и перечислять все такие случаи. Любопытному читателю советуем обратиться к спецификации ”Java Language Specification”.

Теперь мы рассмотрим базовые средства синхронизации в Java.

10.2 Синхронизированные методы и блоки (synchronized)

Мы начнем рассматривать синхронизированные методы и блоки с их синтаксиса, а затем поговорим об особенностях реализации и рассмотрим

[7] Однако, отношение “happens-before” может иметь место и без пессимистичной (блокирующей) синхронизации. Мы убедимся в этом, когда будем рассматривать ключевое слово `volatile`.

[8] Аббревиатура CAS означает “compare-and-swap”, т.е. ”сравнить-и-заменить”. Это разновидность процессорных инструкций, атомарно выполняющих изменение значения в памяти при условии равенства значения указанному. Как правило, такие инструкции используют для реализации неблокирующей синхронизации, речь о которой пойдет далее.

их работу на примере.

Для объявления синхронизированного метода, обычного или статического, достаточно пометить его ключевым словом `synchronized`. Ниже показан пример этого синтаксиса:

```
public class A {

    public synchronized void syncMethod() {
        // тело метода
    }

    public static synchronized void syncStaticMethod() {
        // тело метода
    }

}
```

Синхронизированные блоки по синтаксису схожи с обычными блоками кода, но как и синхронизированные методы, требуют ключевого слова `synchronized`. Для объявления синхронизированного блока потребуется какой-либо объект, в неявном виде предоставляющий монитор для синхронизации блока[9]. Рассмотрим синтаксис таких блоков на простом примере:

```
public class A {

    public Object o = new Object();

    public void methodWithSyncBlock() {
        // в скобках указывается объект для синхронизации
        synchronized (o) {
            // тело блока
        }
    }

}
```

Синхронизированные методы и блоки гарантируют уже знакомое нам отношение “happens-before”. Применительно к ним оно означает, что толь-

[9]О мониторах и их роли в синхронизированных блоках и методах мы поговорим немного позже.

ко один поток в одно и то же время может находиться внутри синхронизированного метода или блока и выполнять код из его тела. Все остальные потоки, пытающиеся выполнить данный метод или блок, ожидают, когда первый поток закончит выполнение тела метода или блока, при любом исходе этого выполнения — в штатном режиме или с выбросом исключения.

Продемонстрируем это на нашем примере со счетчиком, в который мы добавим синхронизированный блок. Код модифицированного примера выглядит так:

```
public class Main {

    private static final int ITERATIONS = 10_000;
    private static Object lock = new Object();
    private static int cnt;

    public static void main(String[] args)
            throws InterruptedException {
        for (int i = 0; i < ITERATIONS; i++) {
            cnt = 0;
            inc();
            if (cnt != 200) {
                System.out.printf("Ошибка. " +
                        "Счетчик оказался равен %d\n", cnt);
            }
        }

        System.out.println("Программа завершена");
    }

    public static void inc()
            throws InterruptedException {
        // оба потока увеличивают счетчик на 100
        Runnable incrementer = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    // синхронизация изменений счетчика
```

```
                    synchronized (lock) {
                        cnt = cnt + 1;
                    }
                }
            }
        };
        Thread t1 = new Thread(incrementer);
        Thread t2 = new Thread(incrementer);
        // стартуем оба потока
        t1.start();
        t2.start();
        // дожидаемся их завершения
        t1.join();
        t2.join();
    }

}
```

В этом примере синхронизированный блок использует объект из поля `lock`, который мы добавили исключительно для целей синхронизации. Но поскольку в синхронизированных блоках можно использовать любой объект, то в этой программе объекта `lock` мы могли бы использовать объект-класс `Main.class` с точно таким же эффектом.

Эта программа является корректной с точки зрения синхронизации памяти. Оба потока захватывают синхронизированный блок (вернее, монитор объекта, но об этом позже) прежде чем прочитать и увеличить текущее значение счетчика. Эту последовательность действий, чтение и изменение значения, единовременно осуществляет только один поток. Поэтому мы могли бы изменить общее количество итераций на существенно большее и конечное значение счетчика всегда было бы равно 200.

Поговорим теперь о синхронизированных методах и блоках более детально. Как мы уже отмечали ранее, их реализация основана на концепции мониторов. Каждый объект в Java неявным образом ассоциирован с монитором, т.е. ресурсом, который может быть захвачен или освобожден каким-либо потоком. Для лучшего понимания принципа работы мониторов можно считать, что монитор эквивалентен ссылке на объект, т.е. разные объекты имеют разные ссылки в памяти, а значит, и мониторы.

Только один поток может захватить монитор в один и тот же момент

времени. Все остальные потоки, пытающиеся захватить монитор, будут заблокированы до момента, пока монитор не будет освобожден. В момент, когда монитор будет освобожден, один из заблокированных потоков будет уведомлен о событии и захватит монитор. Выбор этого потока осуществляется произвольным образом, т.е. Java не дает никаких гарантий порядка с точки зрения выбора следующего потока в момент освобождения монитора.

Интересной особенностью реализации является то, что один и тот же поток может захватить один и тот же монитор несколько раз. В такой ситуации каждое освобождение монитора будет отменять эффект одного действия по его захвату, т.е. в такой ситуации N захватов потребуют N освобождений. Эта логика обеспечивает корректную работу вложенных вызовов синхронизированных методов и вложенных синхронизированных блоков.

При входе в синхронизированный блок или вызове синхронизированного метода текущий поток делает попытку захвата монитора для объекта, ассоциированного с блоком или методом. В этот момент выполнение потока блокируется до успешного захвата монитора. После захвата монитора тело блока или метода выполняется. И, наконец, после завершения выполнения тела блока или метода, независимо от результата этого выполнения — штатного или с выбросом исключения, монитор автоматически освобождается.

Мы уже знаем, что синхронизированные блоки требуют указания объекта для использования его монитора в явном виде. Синхронизированные методы неявным образом используют в качестве такого объекта экземпляр класса, у которого вызван метод. Синхронизированные статические методы используют объект-класс[10] для класса, в котором они объявлены.

10.3 Методы wait(), notify() и notifyAll() класса Object

В базовом классе `java.lang.Object` существует ряд методов, обеспечивающих синхронизацию потоков. Это методы `wait()`, `notify()` и `notifyAll()`. Как и синхронизированные методы и блоки, они используют монитор, ассоциированный с объектом.

Опишем логику работы этих методов:

- `void wait(long timeout)` — приводит к ожиданию наступления ука-

[10]Это знакомые нам по главе про механизм рефлексии объекты класса `Class`.

занного времени или вызова одного из методов `notify()`, `notifyAll()` или `interrupt()` у потока. Для этого метода текущий поток должен являться владельцем монитора, соответствующего объекту данного потока, т.е. должен находиться внутри синхронизированного метода или блока. В противном случае будет выброшено исключение `java.lang.IllegalMonitorStateException`. При вызове метода `wait()` текущий поток освобождает монитор объекта и начинает ожидание. Этот метод также может выбросить знакомое нам исключение `InterruptedException`.

- `void wait()` — аналогичен предыдущему.
- `void notify()` — пробуждает (разблокирует для дальнейшего выполнения) один из потоков, ожидающих монитор объекта, т.е. заблокированных вызовом метода `wait()`. В случае нескольких ожидающих потоков, выбор потока для пробуждения осуществляется произвольно. Пробужденный поток не сможет продолжить выполнение, пока не захватит монитор объекта, а значит, по крайней мере, пока текущий поток не освободит монитор, путем выхода из синхронизированного метода или блока. Как и в случае с методом `wait()`, текущий поток должен являться владельцем монитора, соответствующего объекту данного потока. В противном случае будет выброшено исключение `java.lang.IllegalMonitorStateException`.
- `void notifyAll()` — пробуждает сразу все потоки, ожидающие монитор объекта. В остальном поведение данного метода и пробужденных потоков совпадает с таковыми для метода `notify()`.

С рассмотренными методами связан один подводный камень, удостоенный того, чтобы быть описанным в Javadoc документации JDK. Поскольку механизм `wait()`/`notify()` полагается на низкоуровневые средства, предоставляемые операционной системой, особенности реализации в той или иной ОС влияют на работу этого механизма в Java. Подводный камень заключается в том, что вошедший в ожидание через вызов `wait()` поток может быть пробужден самой операционной системой, а не вызовом `notify()`, `notifyAll()` или `interrupt()`. Такое поведение операционной системы называют "ложным пробуждением" ("spurious wakeup") и встречается оно, как правило, на ОС семейства Linux. Такое пробуждение может нарушить логику программы, поэтому документация рекомендует

дополнять вызовы метода `wait()` проверкой на какое-то дополнительное условие, как правило, меняющееся совместно с вызовом метода `notify()`. Таким образом, рекомендуемый вызов метода `wait()` выглядит примерно так:

```
synchronized (obj) {
    while (<условие-пока-не-наступило>)
        obj.wait();

    // обработка наступления условия
}
```

Отметим, что на практике методы `wait()`, `notify()` и `notifyAll()` используются нечасто. Пример их использования мы увидим в конце этой главы, когда реализуем класс Семафор.

10.4 Ключевое слово volatile

Любое поле в Java, примитивного или объектного типа, обычное или статическое, можно пометить модификатором `volatile`, непосредственно влияющим на синхронизацию потоков при чтении и записи в это поле. Этот модификатор добавляет два свойства для соответствующего поля.

Во-первых, чтение поля и запись в `volatile` поле происходят атомарно. Под атомарной операцией подразумевается, что при совершении такой операции каким-либо потоком ни один другой поток не увидит промежуточного состояния памяти, например, частично считанного или записанного значения примитивного типа. Другими словами, операция будет совершена как единое целое и, в случае ее успешного завершения, результат будет доступен для наблюдения другими потоками полностью.

Это свойство может быть важно для полей типов `long` и `double`, поскольку спецификация[11] допускает выполнение записи в поля этих 64-битных типов как двух последовательных операций записи 32-битных значений. Следовательно, некоторые реализации JVM могут осуществлять запись в поля типов `long` и `double` не атомарно. Модификатор `volatile` применительно к таким полям обеспечивает атомарность записи и чтения.

Во-вторых, между операцией записи в `volatile` поле и последующими операциями чтения из этого поля, происходящими в других потоках, возникает отношение “happens-before”. Это в первую очередь означает, что

[11] Речь идет об уже известной нам спецификации ”Java Language Specification” (JLS).

обновленное значение поля становится доступно другим потокам сразу после его записи.

Отметим важную особенность модификатора `volatile`. Его второе свойство обеспечивает видимость значений `volatile` полей, что существенно отличается от работы синхронизированных методов и блоков. При необходимости совершения нескольких действий модификатора `volatile` для одного или нескольких полей будет недостаточно для обеспечения корректной работы многопоточной программы. Например, если бы мы пометили поле `cnt` модификатором `volatile` в нашем начальном, не синхронизированном примере со счетчиком, то эта программа все равно осталась бы некорректной с точки зрения работы с памятью и конечные значения счетчика могли бы отличаться от 200.

В целом, модификатор `volatile` не является заменой синхронизации, в частности, синхронизированным методам и блокам, и использовать его нужно только при необходимости и к месту. Для корректного использования `volatile` должны соблюдаться следующие условия:

- Запись в поле не должна зависеть от его текущего значения.
- Проверки текущего значения поля не должны включать проверки значений других полей. Примером такой проверки может быть оператор `if` или в условие цикла.

Продемонстрируем теперь пример одного из вариантов практического применения модификатора `volatile`. Мы рассмотрим гипотетическое приложение, которое включает обработку некого сигнала остановки каких-либо вычислений или обработки данных. Условным практическим применением рассматриваемого приема может быть HTTP-сервер, который при получении определенного запроса от администратора может перестать обрабатывать все или часть запросов от пользователей. Итак, рассмотрим код приложения:

```
public class CalcThread extends Thread {

    // флаг остановки вычислений
    private volatile boolean shutdownRequested;

    public void shutdown() {
        shutdownRequested = true;
    }
```

```
    @Override
    public void run() {
        while (!shutdownRequested) {
            // выполняем некие вычисления
            fib(40);
        }
        System.out.println("Вычисления завершены");
    }

    // вычисление чисел Фибоначчи через рекурсию
    // (имеет экспоненциальное время выполнения)
    private int fib(int num) {
        if (num == 0) return 0;
        if (num == 1) return 1;
        return fib(num - 2) + fib(num - 1);
    }

}

public class Main {

    public static void main(String[] args)
            throws InterruptedException {
        System.out.println("Начало вычислений");
        CalcThread thread = new CalcThread();
        thread.start();

        // условное получение сигнала
        // о завершении вычислений
        Thread.sleep(10_000);
        System.out.println("Сигнал остановки");
        thread.shutdown();

        // условное продолжение работы программы
        Thread.sleep(10_000);
```

```
        System.out.println("Программа завершена");
    }

}
```

Это программа выведет в консоль следующее:

```
Начало вычислений
Сигнал остановки
Вычисления завершены
Программа завершена
```

Конечно, конкретно в данном примере мы могли бы использовать механизм кооперативного прерывания работы потоков с помощью знакомого нам метода `interrupt()`. Однако, в случае, если в приложении используется множество потоков и нет прямого доступа к ним, прерывание работы потоков может быть неприемлемо. Кроме того, рассмотренный вариант использования модификатора `volatile` дает больше контроля над точкой остановки вычислений, в то время как сигнал прерывания может привести к выбросу исключения `InterruptedException` в потоках, выполняющих вычисления или обработку данных, что требует корректной обработки прерывания во всех таких потоках.

11 Основные возможности библиотеки java.util.concurrent

Ранее мы рассмотрели базовые средства синхронизации в Java. Однако, стандартная библиотека Java содержит большое число классов, предоставляющих более высокоуровневые абстракции, нежели рассмотренные средства. Эти классы содержатся в пакете `java.util.concurrent`. Данный пакет включает в себя различные утилиты для конкурентного, многопоточного программирования. Подробное рассмотрение всех возможностей, предоставляемых этим пакетом, пожалуй, потребовало бы написание отдельной книги, во многом дублирующей стандартную документацию. Поэтому мы ограничимся перечислением наиболее значимых из этих классов.

Мы отметим следующие подпакеты, классы и интерфейсы из пакета `java.util.concurrent`:

- `ThreadPoolExecutor` — уже знакомый нам класс для управления пулом потоков.

- `ScheduledThreadPoolExecutor` — дочерний класс `ThreadPoolExecutor`, дополняющий возможности родительского класса. Позволяет откладывать выполнение задачи на определенное время или выполнять ее периодически.

- `ConcurrentLinkedQueue` — класс, реализующий потокобезопасную неблокирующую коллекцию-очередь.

- `PriorityBlockingQueue` — класс, реализующий потокобезопасную блокирующую коллекцию-очередь. Эта коллекция отслеживает приоритеты элементов очереди, задаваемые посредством реализации интерфейса `Comparable` или через объект `Comparator`, и имеет методы для изменения очереди, основанные на блокирующей синхронизации.

- `ConcurrentHashMap` — класс, реализующий потокобезопасную коллекцию для ассоциативного массива, основанного на хеш-таблице. Операции чтения элементов в данном классе используют неблокирующую синхронизацию, в то время как операции изменения коллекции подразумевают блокирующую синхронизацию с заданным значением конкурентности. Последнее означает, что хеш-таблица разбивается на некоторое число частей, каждая из которых обновляется независимо и параллельно с другими частями, однако, одновременные изменения в одной и той же части таблицы построены на блокирующей синхронизации и выполняются последовательно.

- `Semaphore` — класс, реализующий семафор. Этот объект ограничивает количество потоков, которым разрешено одновременно войти в определенный участок кода. О семафорах мы поговорим подробно в одном из следующих разделов.

- `CountDownLatch` — класс, позволяющий заблокировать выполнение кода до получения определенного количества сигналов или событий.

- Подпакет `java.util.concurrent.locks` — содержит утилиты для блокировок и ожидания наступления определенного условия. Эти утилиты не повторяют базовые возможности языка, а предоставляют дополнительные возможности. Например, класс `ReentrantLock` помимо возможностей, схожих с синхронизированными блоками, позволяет осуществлять оптимистичную блокировку и может отслеживать очередность ожидающих получения блокировки потоков.

- Подпакет `java.util.concurrent.atomic` — содержит несколько классов, осуществляющих неблокирующую синхронизацию для какого-либо значения примитивного типа или ссылки. Например, класс `AtomicBoolean` позволяет проверить текущее значение булевского типа и, если оно равно переданному ожидаемому значению, изменить его на другое значение в рамках одной атомарной операции[12].

Еще раз отметим, что мы перечислили далеко не все классы рассматриваемого пакета. В частности, он содержит потокобезопасные версии коллекций, списков, ассоциативных массивов и множеств. Кроме этих возможностей, стандартный утилитный класс `Collections` включает в себя набор методов, начинающихся с префикса `synchronized*`, например, `synchronizedSet()` и `synchronizedList()`, позволяющих превратить любую коллекцию в потокобезопасную, посредством блокирующей синхронизации, основанной на использовании одного общего монитора для большинства методов коллекции. Однако, в большинстве случаев потокобезопасные коллекции из пакета `java.util.concurrent` дают большую производительность.

12 Дополнительные возможности многопоточного программирования

Рассматриваемая в этом разделе возможность не является инструментом многопоточного программирования в непосредственном смысле. Скорее, это дополнительная возможность, про которую стоит знать каждому программисту, пишущему многопоточные приложения.

Ранее мы уже говорили о том, что стек потока используется JVM для хранения локальных переменных и других данных, относящихся к выполнению кода потока. Но помимо этих данных в стеке потока допускается хранить так называемые локальные переменные потока (“thread-local variables”). Это значения, инициализируемые, а затем доступные исключительно в рамках выполнения конкретного потока. Другими словами, если поток А сохраняет какое-то значение в локальную переменную потока, то другой поток Б не сможет прочитать ее значение — оно будет доступно только из потока А.

[12]Этот класс, как и многие другие из пакета `java.util.concurrent.atomic`, основан на использовании упоминавшихся ранее CAS-инструкций.

В Java для работы с локальными переменными потока предназначен класс `java.lang.ThreadLocal`. Этот класс параметризован типом локальной переменной потока и содержит следующие методы:

- `T get()` — возвращает значение переменной для текущего потока.
- `void remove()` — очищает значение переменной для текущего потока.
- `void set(T value)` — задает значение переменной для текущего потока.

Локальные переменные потока позволяют ассоциировать какие-то объекты с конкретным потоком без хранения и передачи этих объектов в объект потока в явном виде. На практике это используют, например, для хранения диагностических данных об HTTP-запросах в веб-приложениях (как то, идентификаторах запроса в микросервисных приложениях), что позволяет дополнять этими диагностическими данными записи в логах (журнале) приложения, без необходимости передачи их между модулями.

В качестве демонстрации использования локальных переменных потока мы рассмотрим пример с упрощенной демонстрацией подхода логирования диагностических данных в веб-приложениях. Код нашей программы выглядит следующим образом:

```
public class Main {

    // локальная переменная потока для идентификатора задачи
    private static final ThreadLocal<String> TASK_ID_VAR =
            new ThreadLocal<>();

    public static void main(String[] args)
            throws InterruptedException {
        // создаем пул размера 2
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                2, 2, 60, TimeUnit.SECONDS,
                new LinkedBlockingQueue<Runnable>(5));

        // добавляем в пул задачи
        for (int i = 0; i < 5; i++) {
```

```
            executor.execute(new ThreadLocalRunnable());
        }

        // дожидаемся завершения выполнения всех задач
        executor.shutdown();
        while (!executor.awaitTermination(10,
                TimeUnit.SECONDS)) {
            System.out.println("Ожидаем завершения");
        }
        System.out.println("Все задачи завершены");
    }

    // инициализирует идентификатор и сохраняет его в
    // локальную переменную потока
    public static void initTaskId() {
        // генерируем случайный идентификатор
        UUID id = UUID.randomUUID();
        Main.TASK_ID_VAR.set(id.toString());
    }

    // логирует сообщение + идентификатор
    public static void logWithTaskId(String msg) {
        String taskId = Main.TASK_ID_VAR.get();
        System.out.println(
                String.format("[%s] " + msg, taskId));
    }

}

public class ThreadLocalRunnable implements Runnable {

    @Override
    public void run() {
        initTaskId();

        logWithTaskId("Стартуем задачу");
        try {
```

```
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        logWithTaskId("Задача завершилась");
    }

}
```

В результате выполнения этой программы в консоль будет выведено примерно следующее:

```
[ad594df8-1d4e-4642-bad6-c6956a362c22] Стартуем задачу
[093f2dc2-b42a-4652-96cc-5bfb666da943] Стартуем задачу
[093f2dc2-b42a-4652-96cc-5bfb666da943] Задача завершилась
[ad594df8-1d4e-4642-bad6-c6956a362c22] Задача завершилась
[56179766-9b41-4ad3-b489-df2fa544d2e6] Стартуем задачу
[02795f1c-c4b9-438a-a05c-6924fe18a552] Стартуем задачу
[02795f1c-c4b9-438a-a05c-6924fe18a552] Задача завершилась
[16e36cab-3c83-4331-b61f-dcb2a9b7175e] Стартуем задачу
[56179766-9b41-4ad3-b489-df2fa544d2e6] Задача завершилась
[16e36cab-3c83-4331-b61f-dcb2a9b7175e] Задача завершилась
Все задачи завершены
```

Для каждой из задач, выполняющихся в пуле потоков, генерируется случайный идентификатор, который сохраняется в локальную переменную потока. Пример наглядно показывает, что в дальнейшем для добавления этого идентификатора в лог программы не требуется сохранять этот идентификатор в объекте задачи и передавать его при вызове других модулей. Требуется лишь обратиться к объекту `ThreadLocal` в момент формирования записи лога.

13 Пример класса Семафор

Как мы уже знаем, в стандартной библиотеке есть класс `java.util.concurrent.S` Однако, в учебных целях мы реализуем более простую версию[13] этого класса в рамках этого раздела.

[13]Отметим, что библиотечный класс `Semaphore` устроен по другим принципам и предоставляет гораздо больше возможностей, нежели наша реализация. Рекомендуем интересующимся читателям самостоятельно ознакомиться с этим классом.

Семафор представляет из себя потокобезопасный объект, отслеживающий определенное количество разрешений для конкурирующих друг с другом потоков. Сразу после создания семафора весь его набор разрешений доступен для потоков. Во время дальнейшей работы программы каждый из потоков, использующих семафор, вызывает у объекта-семафора метод получения разрешения и блокируется до получения разрешения. После получения разрешения и завершения связанной с разрешением работы поток обязан освободить разрешение, вызвав соответствующий метод объекта-семафора. Семафоры применяют, когда требуется ограничить одновременное количество потоков, выполняющих некую операцию и/или работающих с каким-то ресурсом.

С точки зрения нашей реализации, отслеживание количества разрешений достаточно тривиально — для этого достаточно использовать счетчик и синхронизировать его чтение и изменение, например, с помощью синхронизированного блока. Однако, оповещение ожидающих потоков, заблокированных до момента освобождения одного из разрешений, является более сложной задачей. Для этой цели мы будем использовать методы `wait()` и `notify()` базового класса `Object`. Отметим также, что наша реализация будет достаточно простой и не будет отслеживать очередь ожидающих потоков, т.е. порядок выдачи разрешений ожидающим потокам никак не гарантирован.

Ниже приведен код нашей программы:

```
/**
 * Класс Семафор
 */
public class Semaphore {

    private final int maxSize;
    private int curSize = 0;

    private final Object lock = new Object();
    private final Object waitLock = new Object();

    public Semaphore(int maxSize) {
        if (maxSize <= 0) {
            throw new IllegalArgumentException("Размерность "
                    + "семафора должна быть положительной");
```

```
    }
    this.maxSize = maxSize;
}

public void acquire() throws InterruptedException {
    boolean needToWait = false;
    synchronized (lock) {
        // пытаемся получить разрешение
        if (curSize < maxSize) {
            curSize++;
            System.out.println("Семафор++: " + curSize);
        } else {
            needToWait = true;
        }
    }

    // если нет свободного места, ждем
    if (needToWait) {
        synchronized (waitLock) {
            waitLock.wait();
        }
        // рекурсия - снова пытаемся получить разрешение
        acquire();
    }
}

public void release() {
    synchronized (lock) {
        if (curSize > 0) {
            curSize--;
            System.out.println("Семафор--: " + curSize);
            // уведомляем один из ожидающих потоков
            // (если они есть)
            synchronized (waitLock) {
                waitLock.notify();
            }
        }
```

```
        }
    }

}

/**
 * Класс "рабочий поток"
 */
public class WorkerThread extends Thread {

    private final int id;
    private final long sleepBeforeReleaseMs;
    private final Semaphore semaphore;

    public WorkerThread(
            int id,
            long sleepBeforeReleaseMs,
            Semaphore semaphore
    ) {
        this.id = id;
        this.sleepBeforeReleaseMs = sleepBeforeReleaseMs;
        this.semaphore = semaphore;
    }

    @Override
    public void run() {
        try {
            log("пытаюсь захватить семафор");
            semaphore.acquire();
            log("успешно захватил семафор");

            log("собираюсь спать в течение "
                    + sleepBeforeReleaseMs + "мс");
            Thread.sleep(sleepBeforeReleaseMs);

            log("собираюсь отпустить семафор");
            semaphore.release();
```

```
            log("отпустил семафор. Конец работы");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private void log(String msg) {
        System.out.println("Поток " + id + ": " + msg);
    }

}

public class Main {

    public static void main(String[] args) {
        final Semaphore semaphore = new Semaphore(2);
        for (int i = 0; i < 4; i++) {
            // берем одинаковое время сна для каждой
            // следующей пары потоков
            int sleepMs = (i / 2 + 1) * 1000;
            WorkerThread thread =
                    new WorkerThread(i, sleepMs, semaphore);
            thread.start();
        }
    }

}
```

В результате работы эта программа выведет в консоль примерно следующее:

```
Поток 0: пытаюсь захватить семафор
Семафор++: 1
Поток 0: успешно захватил семафор
Поток 2: пытаюсь захватить семафор
Поток 3: пытаюсь захватить семафор
Поток 1: пытаюсь захватить семафор
Семафор++: 2
Поток 2: успешно захватил семафор
```

```
Поток 0: собираюсь спать в течение 1000мс
Поток 2: собираюсь спать в течение 2000мс
Поток 0: собираюсь отпустить семафор
Семафор--: 1
Семафор++: 2
Поток 1: успешно захватил семафор
Поток 1: собираюсь спать в течение 1000мс
Поток 0: отпустил семафор. Конец работы
Поток 1: собираюсь отпустить семафор
Поток 2: собираюсь отпустить семафор
Семафор--: 1
Семафор++: 2
Поток 1: отпустил семафор. Конец работы
Семафор--: 1
Поток 2: отпустил семафор. Конец работы
Поток 3: успешно захватил семафор
Поток 3: собираюсь спать в течение 2000мс
Поток 3: собираюсь отпустить семафор
Семафор--: 0
Поток 3: отпустил семафор. Конец работы
```

В целях лучшего освоения материала рекомендуем читателям реализовать эту или аналогичную программу и провести ряд экспериментов.