Многопоточность

Гонка потоков и синхронизация

Содержание

1. Создаем гонку потоков

Сегодня мы рассмотрим самую важную часть многопоточности. Если помните в предыдущей лекции мы скачивали несколько файлов одновременно. Мы создавали обьект класса Thread и передавали туда функциональный интерфейс Runnable в котором запускали наш метод. Давайте упростим себе жизнь и сразу заимплементируем интерфейс в нашем классе скачивания файла.

```
public class DownloadFile implements Runnable {

    private final String url;
    private final String fileName;
    private final int id;

    public DownloadFile(String url, String fileName, int id) {
        this.id = id;
        this.url = url;
        this.fileName = fileName;
    }

    @Override
    public void run() {
        System.out.println(new Date() + "starting " + id);
        try (BufferedInputStream in = new BufferedInputStream(new
URL(url).openStream());
             FileOutputStream fileOutputStream = new
FileOutputStream(fileName)) {
            byte dataBuffer[] = new byte[1024];
            int bytesRead;
            while ((bytesRead = in.read(dataBuffer, 0, 1024)) != -1)
{
                fileOutputStream.write(dataBuffer, 0, bytesRead);
            }
            System.out.println(new Date() + "finished " + id);
        } catch (IOException e) {
            System.out.println(new Date() + "failed " + id);
        }
    }
}
```

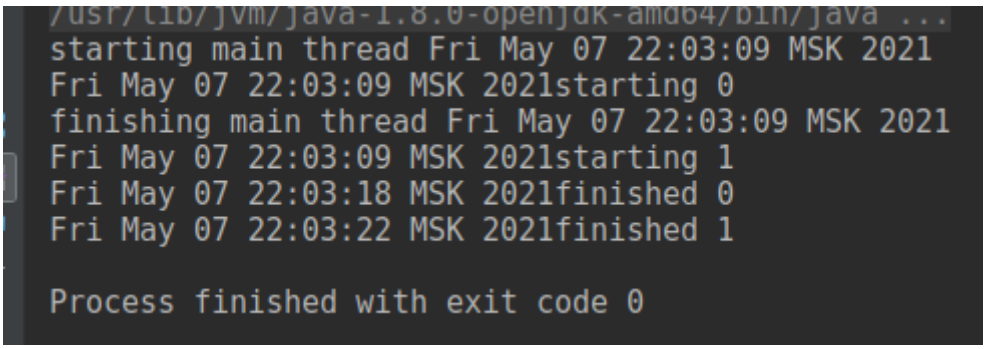
Имплементируем интерфейс Runnable и переопределяем его единственный метод run (потому он и называется функцинальным интерфейсом). Закинем в класс айди чтобы логировать процесс. Когда началось и когда закончилось скачивание файла.

```
public class Main {
    private static final String URL = "https://file-examples-
com.github.io/uploads/2017/04/file_example_MP4_1920_18MG.mp4";

    public static void main(String[] args) {
        System.out.println("starting main thread " + new Date());
        List<Thread> threads = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            threads.add(new Thread(new DownloadFile(URL, i +
"video.mp4", i)));
        }
        for (Thread thread : threads) {
            thread.start();
        }
        System.out.println("finishing main thread " + new Date());
    }
}
```

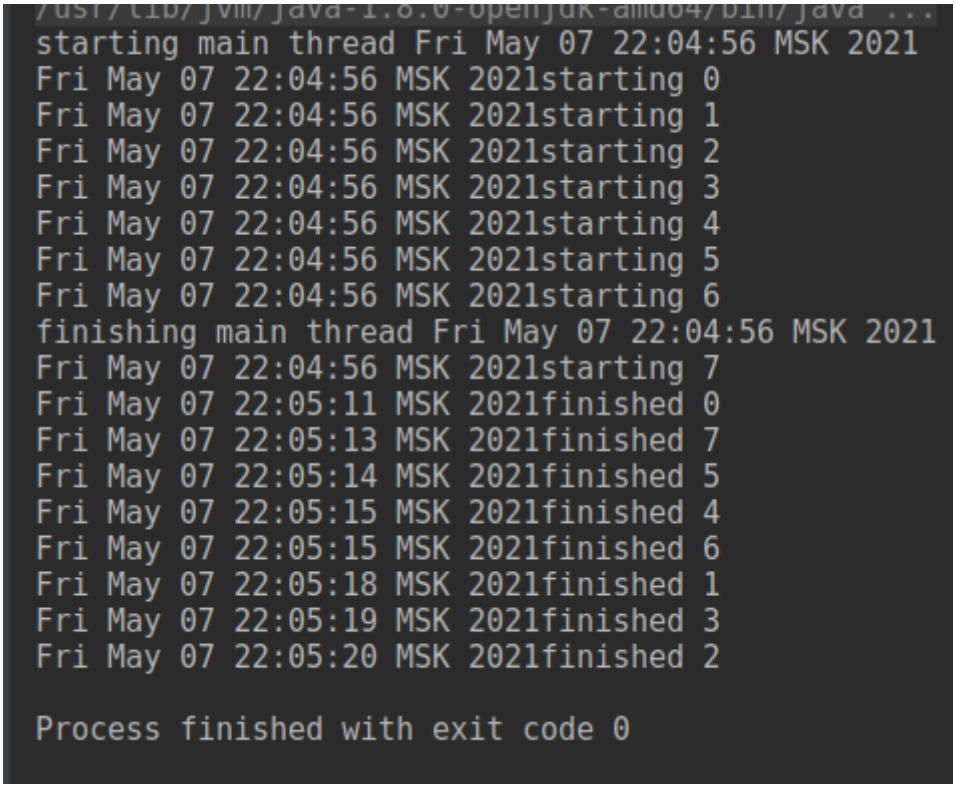
Далее создадим список потоков и запустим их все в цикле. Так же логируем начало и конец нашего главного потока. Да, не забывайте, что по крайней мере тот поток в котором создаете другие тоже начинается и кончается. Теперь, глянем на результаты.

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
starting main thread Fri May 07 21:58:23 MSK 2021
Fri May 07 21:58:24 MSK 2021starting 0
Fri May 07 21:58:24 MSK 2021starting 1
Fri May 07 21:58:24 MSK 2021starting 2
Fri May 07 21:58:24 MSK 2021starting 3
finishing main thread Fri May 07 21:58:24 MSK 2021
Fri May 07 21:58:24 MSK 2021starting 4
Fri May 07 21:58:39 MSK 2021finished 0
Fri May 07 21:58:39 MSK 2021finished 2
Fri May 07 21:58:39 MSK 2021finished 3
Fri May 07 21:58:39 MSK 2021finished 4
Fri May 07 21:58:40 MSK 2021finished 1
```

Итак, сначала стартует наш главный поток. После мы стартуем все потоки для скачивания файлов. Но так как старт потока достаточно тяжелая задача – наш главный поток успевает завершиться. Итого – полное время скачивания 5 файлов – 17 секунд. И здесь мы точно можем сказать что файлы скачивались паралелльно – т.е. независимо друг от друга. Как можно удостовериться в этом? Давайте уменьшим количество потоков до 2.
Как видите – 13 секунд. Что сопоставимо. Здесь мы создали 2 треда и потому меньше ресурсов затратили. Но если бы потоки выполнялись последовательно, т.е. не было бы многопоточности в явном виде, то 5 потоков завершились бы намного дольше.



Для 8 потоков 24 секунды



Но посмотрите что первый поток успел завершиться за 15 секунд. Здесь опять же ситуация что мы создаем слишком много потоков и пока они инициализируется в системе наш главный поток даже успевает перейти к следующей линии кода.

Давайте еще раз прогоровим суть – если у вас стоит задача сделать за минимальное время несколько задач – то многопоточность ваш выход. Но если все задачи можно исполнить независимо друг от друга. Это идеальная ситуация, например для скачивания разных файлов. Или же скачивание большого файла кусками и потом обьединение их в один. Кстати, можно легко узнать когда все потоки завершились – поставить интерфейс в класс и дергать метод завершения – у нас просто пишем в консоль время. И проверять что метод вызывался N раз, просто инкрементируя переменную. Но не все так просто.
Сначала мы рассмотрели способ имплементации интерфейса и передачи обьекта нашего класса классу Thread. Но тогда любопытный человек спросит – а можно сделать проще? Да. Например наследоваться от класса Thread. Если ваш класс еще не наследовался от другого. Давайте рассмотрим этот способ!

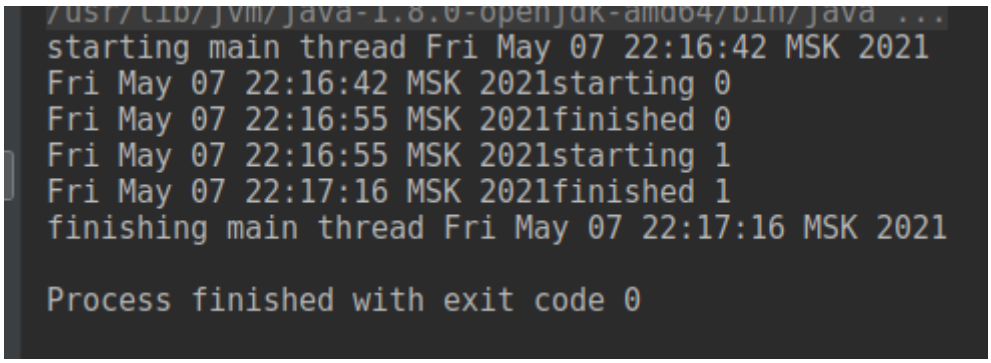
2. Синхронизируем потоки

```
public class DownloadFile extends Thread {

    private final String url;
    private final String fileName;
    private final int id;

    public DownloadFile(String url, String fileName, int id) {
        this.id = id;
        this.url = url;
        this.fileName = fileName;
    }

    @Override
    public synchronized void start() {
        super.start();
        System.out.println(new Date() + "starting " + id);
        try (BufferedInputStream in = new BufferedInputStream(new
URL(url).openStream());
             FileOutputStream fileOutputStream = new
FileOutputStream(fileName)) {
            byte dataBuffer[] = new byte[1024];
            int bytesRead;
            while ((bytesRead = in.read(dataBuffer, 0, 1024)) != -1)
{
                fileOutputStream.write(dataBuffer, 0, bytesRead);
            }
            System.out.println(new Date() + "finished " + id);
        } catch (IOException e) {
            System.out.println(new Date() + "failed " + id);
        }
    }
}
```

Итак, мы наследовались от класс Thread. Пытаемся переопределить его метод start и видим – synchronized void start что за синхронайзд? Что это значит? Почему так?
Ладно, посмотрим что будет если запустить 2 треда в мейне.

```
public static void main(String[] args) {
    System.out.println("starting main thread " + new Date());
    List<Thread> threads = new ArrayList<>();
    for (int i = 0; i < 2; i++) {
        threads.add((new DownloadFile(URL, i + "video.mp4", i)));
    }
    for (Thread thread : threads) {
        thread.start();
    }
    System.out.println("finishing main thread " + new Date());
}
```

Как видите теперь не нужно писать new Thread(new DownloadFile можно сразу класть в список обьект потока. Запустим код!



Эм.. на 2 потока ушло 34 секунды! А не слишком ли много? Посмотрите на логи – стартанул поток 0 потом он завершился и потом стартанул второй поток и завершился и только потом завершился главный поток! Т.е. никакой многопоточности! То же самое если бы мы писали последовательный код. А как же так? Мы же использовали потоки! А все дело именно в этом методе – synchronized void start точнее в ключевом слове. Все обьекты класса будут синхронизироваться. Что это значит? Что пока один поток не завершил свою работу, другой такой же не может быть начат. Т.е. нам для многопоточности нужно выбирать первый способ? Через интерфейс Runnable?
Давайте еще раз – Если у вас независимые задачи и никак друг на друга не влияют, то да.
Но на самом деле можно было переопределить у класса Thread не метод start, а метод run. Вот он не synchronized. Попробуйте сами – переместите код из метода start() в метод run() и запустите мейн.
Давайте разберемся в ключевом слове synchronized

3. Пример использования

Давайте напишем нечто наподобие логера – будем записывать логи приложения.

```
public class Logger {

    private final List<String> list;

    public Logger(List<String> list) {
        this.list = list;
    }

    public void addLog(String header, String body)  {
        list.add("------");
        list.add("Log with header " + header);
        list.add("Log with body " + body);
    }

    public void print() {
        for (String string : list) {
            System.out.println(string);
        }
    }
}
```

Для этого пусть у класса будет список строк и мы будем добавлять в этот список 3 строки. Одну разделяющую и 2 аргумента. Так же напишем метод для вывода всего списка. Заметьте, сначала мы делаем не сихронайзд метод. И посмотрим что выйдет из этого.

В мейне создаем список из потоков, заполняем его и выводим в конце концов через 5 секунд все что было написано в логгер.

```
public static void main(String[] args) {
    Logger logger = new Logger(new ArrayList<>());
    List<Thread> threads = new ArrayList<>();

    for (int i = 0; i < 5; i++) {
        int finalI = i;
        threads.add(new Thread(() ->
                logger.addLog( header: "Header " + finalI,  body: "body " + finalI)));
    }

    for (Thread thread : threads) {
        thread.start();
    }

    TimerTask task = new TimerTask() {
        @Override
        public void run() {
            logger.print();
        }
    };
    Timer timer = new Timer();
    timer.schedule(task,  l: 5000);
}
```

И давайте посмотрим на вывод.

```
------
------
Log with header Header 0
Log with body body 0
Log with header Header 1
Log with body body 1
------
Log with header Header 2
Log with body body 2
------
Log with header Header 4
Log with body body 4
------
Log with header Header 3
Log with body body 3
```

Как видите порядок неправильный. Разделительная линия должна была стоять над заголовком и телом лога. А так как у нас не синхронайзд метод, то у нас гонка потоков. Т.е. в список пишет данные тот поток, который успевает сделать это быстрей всех. Как видите даже порядок нарушен, сначала пишется 4 потом 3.
Как это пофиксить? Добавить синхронайзд к методу addLog. И как видите вывод теперь правильный.

```
public synchronized void addLog(String header, String body)  {
    list.add("------");
    list.add("Log with header " + header);
    list.add("Log with body " + body);
}
```

```
/usr/lib/jvm/java-1.8.0-ope
------
Log with header Header 0
Log with body body 0
------
Log with header Header 1
Log with body body 1
------
Log with header Header 2
Log with body body 2
------
Log with header Header 3
Log with body body 3
------
Log with header Header 4
Log with body body 4
```

На самом деле мы бы могли избежать этого таким путем

```
public void addLog(String header, String body)  {
    list.add(
            "\n------\n" + "Log with header " + header + "\nLog with body " + body
    );
}
```

```

------
Log with header Header 0
Log with body body 0

------
Log with header Header 1
Log with body body 1

------
Log with header Header 2
Log with body body 2

------
Log with header Header 4
Log with body body 4

------
Log with header Header 3
Log with body body 3
```

Да, теперь у нас запись атомарна, т.е. состоит из 1 строки. Но порядок опять же нарушен. Т.е. если вам важен порядок, то пишите синхронайзд метод и старайтесь чтобы в методе был 1 вызов. Если же ну никак не получается этого сделать, то делайте синхронайзд метод всегда.

И давайте в конце поговорим на вот такую тему – зачем нам это знать? Да, скорей всего вы никогда не будете писать действительно многопоточные программы, да и уже есть готовые удобные классы для работы – например CopyOnWriteArrayList который не даст вам совершить ошибку при использовании списка из разных потоков. Но мы должны хотя бы знать о такой штуке как многопоточность, потому как минимум в Андроид у нас будет не 1 поток. Да, мы не будем в андроид запускать много потоков, мы будем запускать другие потоки из основного и их скорей всего будет 1. Но кто знает как вы захотите решить задачу и возможно примените это знание. Да и в любом случае вам нужно знать про ключевое слово synchronized, потому что вы можете наткнуться на него в исходном коде других библиотек.

Для закрепления темы попробуйте написать программу которая запишет все в файл – назовите его log.txt. Так же можете усложнить себе жизнь тем, что проверить количество строк в этом файле и если их больше чем 100 то начинать удалять сверху старые логи. Т.е. в вашем файле не должно быть более 100 линий и это должны быть самые свежие логи. Согласен, сложная задача, но она для тех, кто готов самостоятельно разобраться в этом вопросе.