

Практика 1: Введение, потоки

16.02.2018г

1. Формальности

Итак, начинается вторая часть изучения программирования на примере языка Java. Пары в этом семестре будут логическим продолжением того, что было в предыдущем семестре — немножко больше многопоточности, программирования сетевых приложений, многопоточных И сетевых приложений, приложений с пользовательским интерфейсом наконец-то, и даже немного веб-приложений, если мы до них дойдём. Как я понял, так или иначе почти всё затрагивалось при программировании под android, но в этот раз всё будет более системным и обстоятельным.

Сначала, как водится в начале семестра, формальности. В конце будет экзамен, оценка за который получается, в том числе, и из оценки по практике, так что в этом семестре будет не просто зачёт/незачёт, а некоторое число. Чтобы получить оную оценку, надо, как обычно, качественно и вовремя делать домашки, писать контрольные, сдавать их через HwProj (<http://hwproj.me/courses/26>), делая пуллреквест в свой репозиторий и кидая в HwProj ссылку на пуллреквест. В HwProj надо записаться на второй семестр этого курса. Домашек будет меньше, чем в прошлом семестре, но они будут более объёмными. Списывать, как обычно, нельзя, и вообще, имеет смысл стараться решать задачи самостоятельно, курс ориентирован на сольное прохождение, к тому же, если что — всегда можно спросить у меня.

Пары у нас будут один раз в две недели, поэтому будет меньше времени практиковаться, но домашек особо меньше не будет. На самом деле, на парах я в основном буду что-то рассказывать.

Напомню про то, за что снимались баллы в прошлом семестре:

Пропущенный дедлайн	баллы делятся на два
Задача на момент дедлайна не реализует все требования условия	пропорционально объёму невыполненных требований
Неумение пользоваться гитом	-2
Проблемы со сборкой (в том числе, забытый <code>org.jetbrains.annotations</code>)	-2
Отсутствие JavaDoc-ов для всех классов, интерфейсов и публик-методов	-2
Отсутствие описания метода в целом	-1
Слишком широкие области видимости для полей	-2
<code>if (...) return true; else return false;</code>	-2
Именование классов-полей-методов-... и прочие <code>code conversions</code>	-1
Неиспользование <code>try-with-resources</code> там, где это было бы уместно	-1
Комментарии для параметров с заглавной буквы	-0.5

Обнаружение ошибок из этого списка сразу влечёт снятие баллов за задачу, даже без права исправить ошибку. На остальные ошибки я буду указывать и будет возможность их поправить, но наиболее распространённые ошибки будут пополнять этот список, так что имеет смысл ходить на пары, чтобы вовремя узнать, что меня в очередной раз ужаснуло и заставило расширить список “плохих” ошибок. Табличку с оценками я выложу на вики.

2. Многопоточность

Теперь перейдём к содержательной части пары. На теории вы уже должны были начать потоки, поэтому я ещё немного про них расскажу и немного попрактикуемся (а в домашке попрактикуетесь много).

Вообще многопоточные программы надо уметь писать, потмоу что нынче даже в телефонах процессоры имеют по четыре ядра, не говоря уже о настольных компьютерах и ноутбуках. Сильно грузить одно ядро, чтобы при этом остальные простаивали — страшный грех нынче. Но дело не только в высокопроизводительных вычислениях (которые всё-таки не так часто встречаются) — многопоточные программы позволяют пользователю продолжать взаимодействовать с интерфейсом и иметь возможность узнать прогресс длительной операции, или вообще её отменить. При этом длительная операция не обязательно длительная, потому что её считать сложно — нет, это может быть, например, сетевой запрос, инициализация какого-нибудь устройства, да даже просто загрузка большого файла (или набора файлов, что часто приходится делать, например, в играх). Такие операции вообще не грузят процессор, поэтому, кстати, их можно запускать несколько сразу одновременно, чем пользуются, например, торрент-клиенты — передаваемый файл разделяется на кучу маленьких кусков, каждый из которых качается независимо в отдельном потоке. Игры грузят много файлов тоже как правило в несколько потоков одновременно. Есть ещё чисто практическое соображение в пользу многопоточности — на любом уважающем себя собеседовании про многопоточность будут вопросы.

Но иногда неопытные программисты, узнав о потоках, начинают вставлять их повсюду, что на самом деле приводит к разрушительным последствиям. Многопоточные программы гораздо сложнее в отладке, могут содержать сложнообнаружимые баги и вообще отличаются непредсказуемостью поведения. К тому же, не факт, что многопоточная программа будет работать быстрее такой же однопоточной — переключение между потоками имеет свою цену с точки зрения производительности, а неграмотное использование потоков может заставить процессор тратить 95% рабочего времени на обслуживание потоков. Так что много потоков — не панацея, и надо такие программы писать очень аккуратно.

Немного про способы прострелить себе ногу: концептуально их всего два, состояние гонки (race condition) и взаимная блокировка потоков (deadlock).

3. Состояние гонки

Состояние гонки — это любая ситуация, когда результат работы программы зависит от переключения потоков планировщиком. Эта ситуация во всех практических случаях считается ошибкой, потому что предсказать поведение планировщика нельзя и внешне всё выглядит так, будто результат работы программы случаен. Типичный пример гонки показан на картинке:



Положим, что два потока одновременно хотят обновить баланс. Если планировщик сначала даст поработать первому потоку, тот считает значение текущего баланса, тут может так случиться, что планировщик остановит его и даст поработать второму потоку. Второй поток тоже считает текущее значение баланса (то же, его ведь ещё не обновили), прибавит к нему 200 рублей и сохранит результат, в итоге обновлённый баланс будет 1200 рублей. Тут планировщик видит, что поток 2 завершён и даст доработать потоку 1, который прибавит 100 рублей к считанной им ранее сумме баланса, получив 1100 рублей, и радостно сохранит эти самые 1100 рублей в качестве текущего баланса. 200 рублей потеряно. А могло получиться и наоборот, и потерялись бы 100 рублей, что хотя и вдвое меньше, но тоже не очень.

Такая ситуация может быть слишком редкой, чтобы её надёжно воспроизвести, но в этом и проблема с гонками — программа может работать правильно большую часть времени, примерно раз в 10 лет теряя одну транзакцию. Вот пример программы, которая, скорее всего, будет работать странно достаточно часто:

```
int[] a = new int[1000];
for (int i = 0; i < a.length; ++i) {
    a[i] = 1;
}
```

```

int[] result = new int[1];

for (int i = 0; i < 100; ++i) {
    final int localI = i;
    new Thread(() -> {
        for (int j = localI * 10; j <= (localI + 1) * 10 - 1; ++j) {
            result[0] += a[j];
        }
    }).start();
}

Thread.sleep(100);

System.out.println("Result = " + result[0]);

```

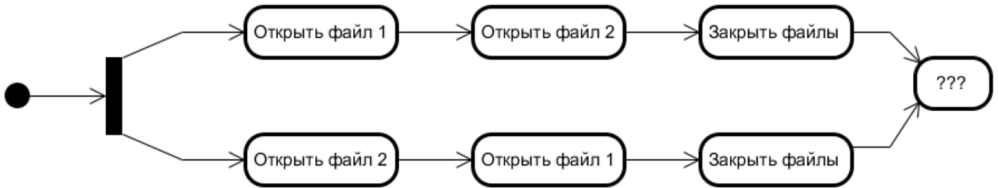
Берём массив из тысячи элементов, заполняем его единицами, и хотим посчитать сумму чисел, которые в нём лежат, раз уж мы умные, в 100 потоков. Для этого делим массив на одинаковые куски по 10 элементов и суммируем каждый кусок в отдельном потоке, обновляя общую для всех потоков переменную с результатом. Обратите внимание на странные приёмы, которые тут используются — переменная `localI` заведена, чтобы в замыкание в лямбда-функцию, которая будет работать в отдельном потоке, попало именно текущее значение `i`, а не счётчик цикла (иначе это ошибка компиляции), а массив из одного элемента — это бюджетный способ получить мутабельное `int`-овое значение на куче. Так в продакшн-коде лучше не делать, но зато на слайд помещается. Ещё мы тут не пытаемся дождаться выполнения всех потоков, а просто надеемся, что они завершатся за 100 миллисекунд, так тоже в продакшн-коде делать нельзя (впрочем, тут, скорее всего, большинство потоков успеет завершиться до того, как стартуют следующие за ними, благо считать каждому потоку особо нечего).

Этот пример можно запустить и посмотреть, что будет. Иногда получается 1000, иногда 999, а иногда и 992. Если мы сделаем более грамотно и сначала создадим все потоки, а потом всех их одновременно запустим, то по идее должно быть ещё хуже. Почему получается что-то странное? Операция “+” не атомарна, так что получается как на картинке — один поток считал значение `result[0]`, второй поток считал то же значение, прибавил, записал, первый поток тоже прибавил и тоже записал, затерев изменения второго.

Обратите внимание, что эта ситуация не специфична для Java, процессоров Intel или ещё чего-нибудь, так можно испортить себе программу где угодно. Бывают и более странные случаи гонок, о которых вам либо уже рассказали, либо расскажут на лекциях, связанные с перестановкой инструкций на конвейере процессора, наличием в процессоре `write buffer`-ов и т.д. А ещё компилятор имеет право менять порядок вычислений в целях оптимизации, и делает он это без учёта того, что другие потоки могут использовать промежуточные результаты этих вычислений.

4. Deadlock

Второй способ прострелить себе ногу — когда поток ждёт наступления какого-нибудь события, которое наступит, только если поток продолжит работу, чего он сделать не может, поскольку ждёт наступления этого события. Например, когда двум потокам нужно два файла, чтобы работать, первый поток успел открыть первый файл, второй поток успел открыть второй, и оба ждут, пока другой нужный им файл освободится. При этом ни один поток не может продвинуться, поэтому не может и освободить файл:



Самый простой способ добиться этой ситуации в программе — это

```
Thread.currentThread().join();
```

— поток ждёт, пока он же не завершится, но он не может завершиться, потому что он ждёт. Естественно, это игрушечный пример, но в реальной жизни заблокироваться на самом себе тоже совсем несложно — например, неаккуратная работа с рекурсией (к счастью, встроенные в Java механизмы синхронизации с рекурсией работать умеют), хитрые схемы многопоточной обработки событий, когда один поток послал сигнал и ждёт на него ответа, чтобы получить который он должен ответить на какой-то другой сигнал.

Хорошая новость в том, что блокировка потока в принципе возможна только если выполняются одновременно четыре условия:

- потокам нужен доступ к неразделяемому ресурсу, то есть ресурсу, владеть которым в каждый момент времени может только один поток;
- поток удерживает хотя бы один ресурс, ожидая доступа к другому ресурсу;
- у потока нельзя “отнять” ресурс, то есть он может освободить его только сам;
- каждый поток ожидает ресурса, занятого каким-то другим потоком, то есть имеется цикл зависимости по ресурсам.

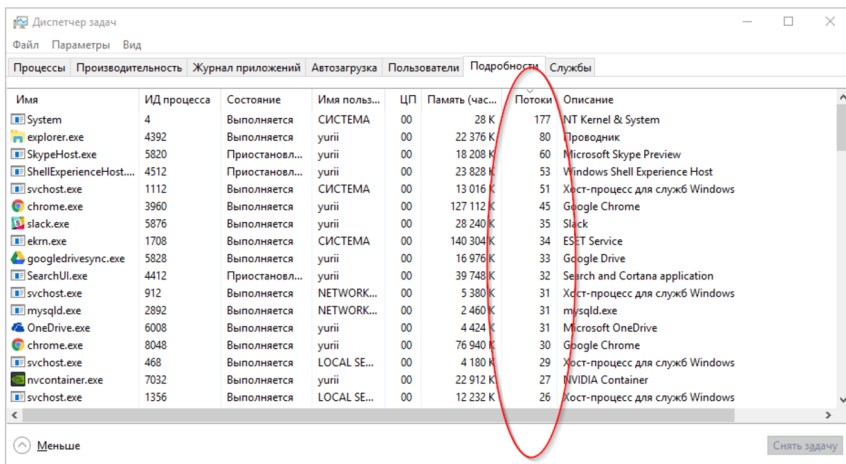
Если хоть одно из этих условий не выполнено, то блокировка в принципе невозможна, поэтому проблему с блокировками можно решить, делая невозможным наступление одного из четырёх этих условий. Lock-free-подход — “нет неразделяемых ресурсов — нет проблем”, подход, реализованный в C++-ном `std::lock` — “если лочить ресурсы всегда в одинаковом порядке, то не будет циклов” и т.д. У нас же ресурсами будут, как правило, критические секции кода, они по определению неразделяемы, и надо внимательно следить, чтобы не пытаться синхронизироваться сразу по нескольким разделяемым переменным в неправильном порядке.

5. Поток в Windows

Потоки в Java не обязаны отображаться напрямую в потоки операционной системы, но тем не менее в стандартной реализации отображаются. Так что понимать, что такое поток в операционной системе, было бы нелишне (хотя, конечно, это должны рассказывать на курсах про ОС, напомнить не помешает). Конкретно в Windows поток представляет собой некоторое количество записей в пространстве ядра и в пользовательском адресном пространстве, а именно:

- Thread Kernel Object (~1240 байт)
- Thread environment block (TEB) (4 Кб)
- User-mode stack (1 Мб)
- Kernel-mode stack (24 Кб)

Потоку, как только он создаётся, сразу выделяется 1 мегабайт оперативки под стек, и поток может попросить у ОС увеличить размеры стека. Казалось бы, мелочь, но вспомните пример про сложение единиц в массиве, там мы создали 100 потоков, минус 100 мегабайт оперативной памяти сразу. Тоже, казалось бы, мелочь, но вот скриншот диспетчера задач моего ноутбука сразу после старта винды:



Имя	ID процесса	Состояние	Имя польз...	ЦП	Память (час...	Потоки	Описание
System	4	Выполняется	СИСТЕМА	00	28 K	177	NT Kernel & System
explorer.exe	4392	Выполняется	yurii	00	22 376 K	80	Проводник
SkypeHost.exe	5820	Приостанов...	yurii	00	18 208 K	60	Microsoft Skype Preview
ShellExperienceHost...	4512	Приостанов...	yurii	00	23 828 K	53	Windows Shell Experience Host
svchost.exe	1112	Выполняется	СИСТЕМА	00	13 016 K	51	Хост-процесс для служб Windows
chrome.exe	3960	Выполняется	yurii	00	127 112 K	45	Google Chrome
slack.exe	5876	Выполняется	yurii	00	28 240 K	35	Slack
ekm.exe	1708	Выполняется	СИСТЕМА	00	140 304 K	34	ESET Service
googledrivesync.exe	5828	Выполняется	yurii	00	16 976 K	33	Google Drive
SearchUI.exe	4412	Приостанов...	yurii	00	39 748 K	32	Search and Cortana application
svchost.exe	912	Выполняется	NETWORK...	00	5 380 K	31	Хост-процесс для служб Windows
mysqld.exe	2892	Выполняется	NETWORK...	00	2 460 K	31	mysqld.exe
OneDrive.exe	6008	Выполняется	yurii	00	4 424 K	31	Microsoft OneDrive
chrome.exe	8048	Выполняется	yurii	00	76 940 K	30	Google Chrome
svchost.exe	468	Выполняется	LOCAL SE...	00	4 180 K	29	Хост-процесс для служб Windows
nvcontainer.exe	7032	Выполняется	yurii	00	22 912 K	27	NVIDIA Container
svchost.exe	1356	Выполняется	LOCAL SE...	00	12 232 K	26	Хост-процесс для служб Windows

Все эти потоки большую часть времени проводят в спящем режиме, так что процессорного времени не требуют, но оперативку под стек каждому потоку всё-таки надо. А их тысячи. Ещё, кстати, для каждой загруженной в приложение нативной разделяемой библиотеки (включая kernel32.dll, user32.dll, msvcrt32.dll и прочие подобные вещи) при запуске *каждого* потока вызывается функция DllMain с параметрами DLL_THREAD_ATTACH и при завершении *каждого* потока — DLL_THREAD_DETACH. Библиотека вправе на эти параметры не реагировать (и большинство так и делают), но вызвана функция всё равно будет. Обычное нативное приложение грузит десятки разделяемых библиотек (свои, фреймворка, на котором оно написано, ядра Windows), так что это удар ещё и по производительности (к счастью, Java и .NET не используют эту чудо-систему).

Поэтому желательно проектировать программу не как набор потоков, а как набор *задач*, каждая из которых может выполняться в отдельном потоке, но отработывает быстро, поэтому потоки можно переиспользовать. Пулы потоков, которые реализованы в любой нормальной стандартной библиотеке современных языков, в том числе Java, как раз для поддержки такой парадигмы.

6. Задача на дом

Теперь про условие домашней задачи (на три недели, она относительно большая):

- нужно реализовать класс `ThreadPoolImpl`, представляющий собой пул задач с фиксированным числом потоков (число задается в конструкторе);
- при создании объекта `ThreadPoolImpl` в нём должно начать работу n потоков;
- у каждого потока есть два состояния: ожидание задачи и выполнение задачи;
- задача — вычисление некоторого значения, вызов `get` у объекта типа *Supplier<R>*;
- при добавлении задачи, если в пуле есть ожидающий поток, то он должен приступить к ее исполнению. Иначе задача будет ожидать исполнения, пока не освободится какой-нибудь поток;
- задачи, принятые к исполнению, представлены в виде объектов интерфейса `LightFuture`, про который подробнее дальше;
- у `ThreadPoolImpl` должен быть метод `shutdown`, который завершает работу всех потоков (через `Thread.interrupt()`).

`LightFuture` — ссылка на значение, которое станет доступно когда-то в будущем. Это должен быть генерик (параметризованный типом возвращаемого значения), имеющий следующие методы:

- `isReady`, который возвращает `true`, если задача выполнена;
- `get`, который возвращает результат выполнения задачи или блокирует вызывающего до тех пор, пока задача не будет выполнена;
 - в случае, если соответствующий задаче `supplier` завершился с исключением, этот метод должен завершиться с исключением `LightExecutionException`;
- `thenApply` — принимает объект типа `Function`, который может быть применен к результату данной задачи X и возвращает новую задачу Y , принятую к исполнению. При этом:
 - новая задача будет исполнена не ранее, чем завершится исходная;
 - в качестве аргумента объекту `Function` будет передан результат исходной задачи, и все Y должны исполняться на общих основаниях (т.е. должны разделяться между потоками пула);

- метод `thenApply` может быть вызван несколько раз;
- метод `thenApply` не должен блокировать работу потока, если результат задачи `X` ещё не вычислен.

Использование такого пула потоков может выглядеть, например, вот так:

```
ThreadPoolImpl<Integer> pool = new ThreadPoolImpl<>(5);
LightFuture<Integer> task = pool.addTask(() -> 2 * 2);
assertThat(task.get(), is(4));

LightFuture<Integer> task1 = pool.addTask(() -> 2 * 3);
LightFuture<Integer> task2 = task1.thenApply((i) -> i + 1);
LightFuture<Integer> task3 = task1.thenApply((i) -> i + 2);
assertThat(task1.get(), is(6));
assertThat(task2.get(), is(7));
assertThat(task3.get(), is(8));
```

Естественно, возможны варианты, например, `LightFuture` можно сделать вложенным классом `ThreadPoolImpl`-а.

Кроме этого, ещё хочется следующего:

- не использовать пакет `java.util.concurrent`, потому что там это уже реализовано, причём по-настоящему;
- все интерфейсные методы должны быть потокобезопасны, то есть и пул потоков, и `LightFuture` сами могут использоваться из нескольких потоков одновременно;
- для каждого базового сценария использования должен быть написан тест;
- обязателен билд в CI, на котором проходят ваши тесты;
- дедлайн: до **10:00 09.03.2018**.

7. Задача на остаток пары

Реализовать интерфейс, представляющий ленивое вычисление:

```
public interface Lazy<T> {
    T get();
}
```

Есть некоторое вычисление, представляемое объектом `Supplier<T>`, его должно быть можно обернуть в реализацию `Lazy<T>`, которая выполнит вычисление при первом обращении к методу `get()`. При последующих обращениях к методу `get()` должен возвращаться уже посчитанный результат, `Supplier` второй раз вызываться не должен (вычисление вообще не должно производиться более одного раза). То есть отработавший `Supplier` становится больше не нужен в `Lazy` и его можно смело удалить.

Поскольку это задача на многопоточность, реализаций должно быть две:

- однопоточная реализация, гарантирующая корректную работу, если поток всего один;
- многопоточная реализация, гарантирующая корректную, но несколько более медленную работу в случае, когда с *Lazy* могут быть обращения из нескольких потоков сразу.

А поскольку у нас есть две реализации одного интерфейса, создавать их надо не вручную, а сделав класс *LazyFactory*, у которого есть два статических метода вида *public static <T> Lazy<T> create...Lazy(Supplier<T>)*, возвращающих либо первую реализацию, либо вторую.

При этом:

- каждый объект *Lazy* должен иметь не больше двух полей (обратите внимание, *объект*, статические поля допустимы);
- *Supplier.get* вправе вернуть *null*;
- должны быть тесты:
 - однопоточные, на разные хорошие и плохие случаи;
 - многопоточные, на наличие гонок.

Эту задачу надо доделать дома и сдать тоже через HwProj (дедлайн такой же, как у предыдущей задачи, до **10:00 09.03.2018**).