Практика 1: Введение, потоки

Юрий Литвинов
yurii.litvinov@gmail.com

16.02.2018г

Правила игры

- Пара один раз в две недели
  - $=>$ вдвое больше домашки за один раз
- Как обычно, куча домашек, две контрольные (и переписывание в конце), баллы и дедлайны, HwProj
  - Задач будет немного меньше, но они немного объёмнее
- За практику будет выставляться оценка, которая потом будет учитываться при сдаче экзамена
  - по какой-то хитрой формуле, учитывающей баллы за домашку и контрольные
  - Максимальный итоговый балл: 1.5, складывается из балла за контрольные (макс. 0.75) и балла за домашки (макс. 0.75). Максимум за к/р — 16 баллов, максимум за д/з пока не известен
- Будет про многопоточные, сетевые, сетевые И многопоточные приложения, пользовательский интерфейс и т.д.

Напоминание про штрафы

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 org.jetbrains.annotations)	-2
Отсутствие JavaDoc-ов для всех классов, интерфейсов и паблик-методов	-2
Отсутствие описания метода в целом	-1
Слишком широкие области видимости для полей	-2
if (...) return true; else return false;	-2
Именование классов-полей-методов-... и прочие code convertions	-1
Неиспользование try-with-resources там, где это было бы уместно	-1
Комментарии для параметров с заглавной буквы	-0.5

Список может расширяться!

Многопоточное программирование вообще

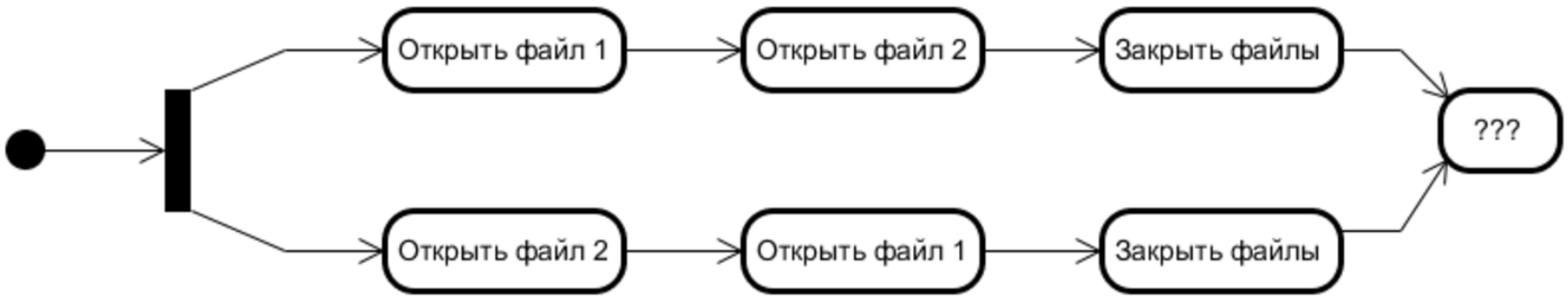
- Плюсы
  - Не вешать пользовательский интерфейс
  - Равномерно распределять вычислительно сложные задачи по ядрам
  - Выполнять одновременно несколько блокирующих операций ввода-вывода
- Минусы
  - Тысяча способов прострелить себе ногу
  - Не всегда многопоточная программа работает быстрее однопоточной

Race condition

Маленький пример на race condition

```
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]);
```

Deadlock


Открыть файл 1
Открыть файл 2
Закрыть файлы
???
Открыть файл 2
Открыть файл 1
Закрыть файлы

Очень маленький пример на deadlock

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

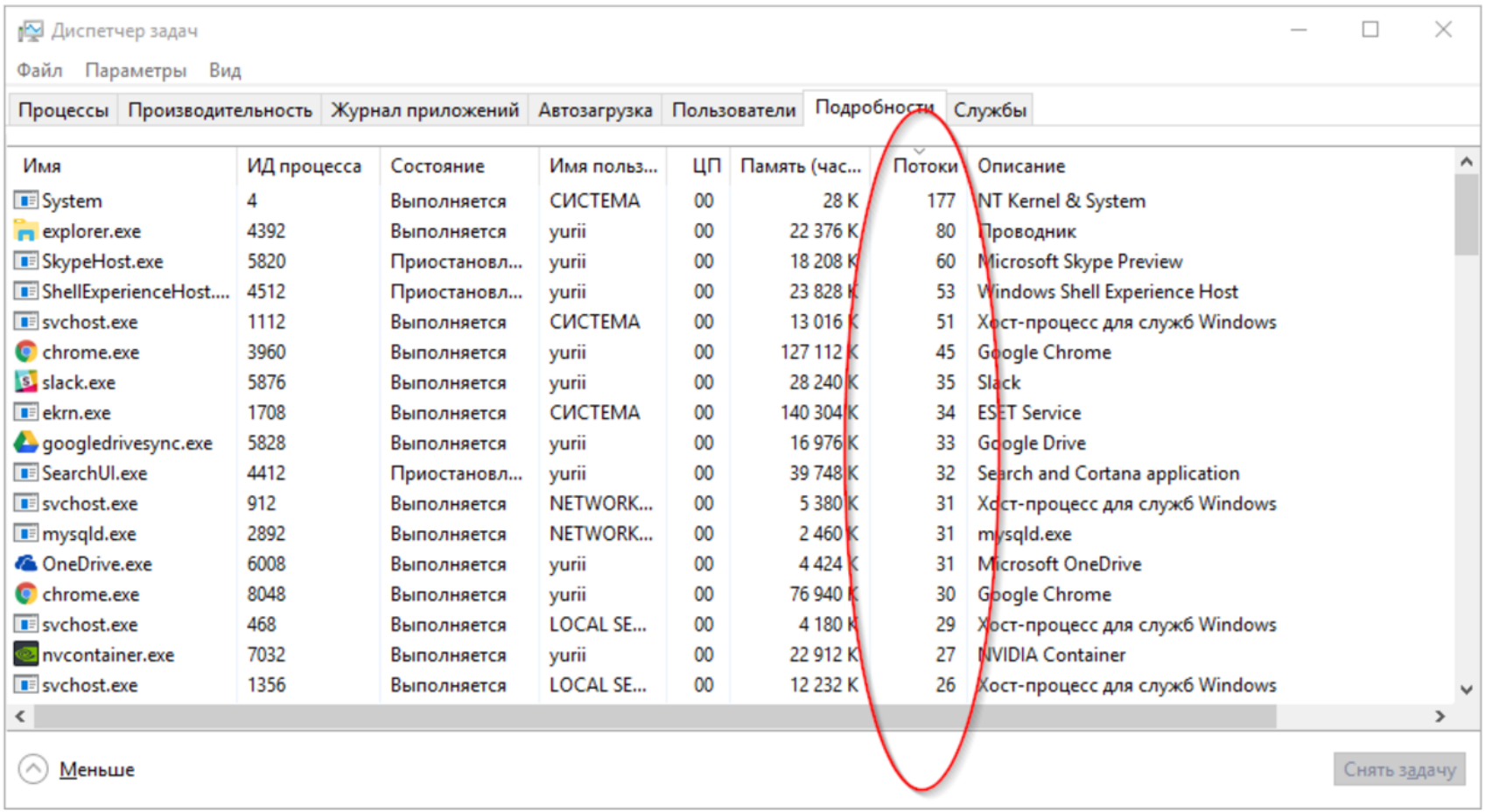
Пример, потоки в Windows

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

Ещё для каждой dll-ки, загруженной для процесса при старте или остановке потока вызывается DllMain с параметрами DLL_THREAD_ATTACH и DLL_THREAD_DETACH

Квант времени — 20-30 мс, после чего происходит *переключение контекстов*

Как делать не надо



Задача на дом (пул потоков)

- Нужно реализовать простой пул задач с фиксированным числом потоков (число задается в конструкторе)
- При создании объекта *ThreadPoolImpl* в нем 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*
- Метод *shutdown* должен завершить работу потоков (через *Thread.interrupt()*)

LightFuture

- Метод *isReady* возвращает *true*, если задача выполнена
- Метод *get* возвращает результат выполнения задачи
  - В случае, если соответствующий задаче *supplier* завершился с исключением, этот метод должен завершиться с исключением *LightExecutionException*
  - Если результат еще не вычислен, метод ожидает его и возвращает полученное значение

LightFuture

- Метод *thenApply* — принимает объект типа *Function*, который может быть применен к результату данной задачи *X* и возвращает новую задачу *Y*, принятую к исполнению
  - 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

Примеры использования

```
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));
```

Примечания

- В данной работе запрещено использование содержимого пакета *java.util.concurrent*
- Все интерфейсные методы должны быть потокобезопасны
- Для каждого базового сценария использования должен быть написан несложный тест
- Обязателен билд в CI, на котором проходят ваши тесты
- Дедлайн: до **10:00 09.03.2018**

Задача на пару, многопоточный Lazy

Реализовать следующий интерфейс, представляющий ленивое вычисление:

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

- Объект *Lazy* создаётся на основе вычисления (представляемого объектом *Supplier*)
- Первый вызов *get()* вызывает вычисление и возвращает результат
- Повторные вызовы *get()* возвращают **тот же** объект, что и первый вызов
- Вычисление должно запускаться не более одного раза

LazyFactory

Создавать объекты надо не вручную, а с помощью класса *LazyFactory*, который должен иметь два метода с сигнатурами вида *public static <T> Lazy<T> create...Lazy(Supplier<T>)*, возвращающих две разные реализации *Lazy<T>*:

- Простая версия с гарантией корректной работы в однопоточном режиме (без синхронизации)
- Гарантия корректной работы в многопоточном режиме; вычисление не должно производиться более одного раза
  - Что-то наподобие многопоточного синглтона

При этом

- Ограничение по памяти на каждый *Lazy*-объект: не больше двух ссылок
- *Supplier.get* вправе вернуть *null*
- Тесты
  - Однопоточные, на разные хорошие и плохие случаи
  - Многопоточные, на наличие гонок
- Доделать дома