Параллельное программирование: Лекция/Практика 3

Учебный курс

Кензин Максим Юрьевич
Кафедра Информационных технологий
ИМИТ ИГУ / Институт динамики
систем и теории управления СО РАН

ИМИТ ИГУ, 2022

Посмотрели

- Как создать поток:
  - Extends Thread;
  - Implements Runnable.
- Базовые методы класса поток;
- Методы, вызывающие вход потока в режим ожидание:
  - Sleep();
  - Join();
- Использование конструкции try-catch-finally;
- Прерывание потока «снаружи» и «изнутри».

«Скромный» поток

1. **Метод yield() информирует планировщика потоков, что текущий поток готов отказаться от своего текущего использования процессора, но хотел бы, чтобы его запланировали как можно скорее (чаще всего поток перемещается в низ очереди потоков равного приоритета).**

1=2=3

1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2

1>2>3

1	2	1	2	1	3	1	2	1	2	1	3	1	2	1	2	1	3	1	2

2. **Используется редко, так как обладает недетерминированным поведением, в отличие от** *sleep, join, wait***.**
3. **Метод может быть полезным, например, когда поток ожидает наступления какого-либо события и необходимо чтобы проверка его наступления происходила как можно чаще.**

Служебный поток

Поток можно определить как служебный (Daemon/поток-демон), тогда он будет завершаться «насильно» после завершения всех обычных пользовательских потоков (User threads).

Порядок!

```
NewThread Thread = new NewThread();
Thread.setDaemon(true);
Thread.run();
```

1. **Все потоки в main{} по умолчанию являются пользовательскими;**
2. **Свойство служебности потока проверяется как** `Thread.isDaemon()` **и наследуется при создании новых потоков.**
3. **Из-за «насильного» завершения потоков целостность данных не гарантируется.**

Синхронизация. Память.

1. **Память – глобальная и встроенная в процессор (регистры и кэш).**
2. **В кэше хранятся самые часто используемые переменные (JVM сама управляет и решает какие именно) для быстрого доступа к ним. У каждого потока свой кэш.**
3. **Если одна переменная должна использоваться несколькими потоками, может оказаться, что каждый поток работает со своей локальной копией этой переменной в кэше.**

```
public volatile int Index = 0;
```

4. **Volatile гарантирует:**
   - **безопасные чтение и запись переменной (даже не унарные *long* и *double*), но не изменение;**
   - **Не_помещение переменной в кэш потоков.**

Синхронизация. Мьютексы.

1. **Мьютекс (от англ. «mutex», «mutual exclusion» — «взаимное исключение») – механизм ограничения доступа к объекту.**

2. **Мьютекс принимает два значения Занят/Свободен.**

3. **Мьютекс прикреплен к каждому объекту в Java, но доступ к нему есть только у JVM. От программиста доступ скрыт.**

4. **Но! Можно работать с мьютексом посредством монитора (надстройка над мьютексом). В Java монитор –** `synchronized`.

Синхронизация

1. **Чтобы запретить одновременный доступ к данным нескольких потоков, используем оператор synchronized(){}.**

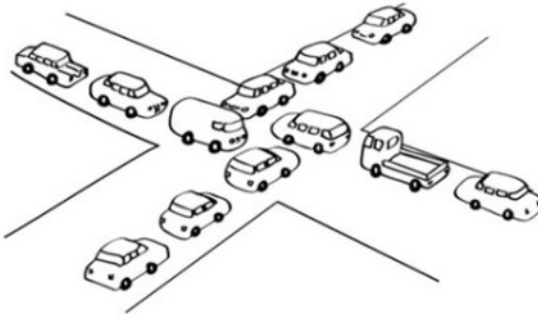
```
public void swap() {
        //Работа нескольких потоков
        synchronized (object) {
        //...логика метода
        }
}
```

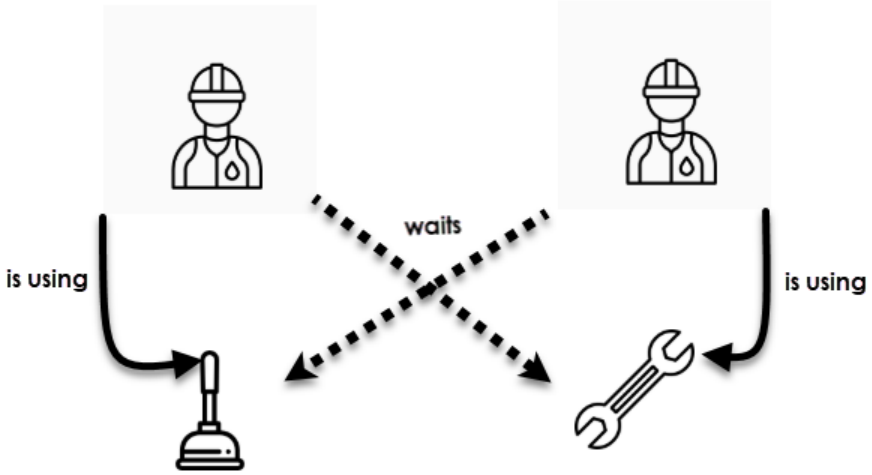
```
public synchronized void swap() {
        //...логика метода
}
```

2. **В качестве `object` нельзя использовать примитивный тип.**
3. **Синхронизация по объекту запрещает доступ остальным потокам в синхронизированный блок кода.**
4. **Синхронизация по методу запрещает доступ другим потокам как в сам синхронизированный метод, так и в другие методы класса с синхронизацией.**
5. **Аккуратно следим, кто и когда отдает полученный лок.**

Синхронизация. Deadlock.

1. **Deadlock (взаимная блокировка) — это ошибка, которая происходит когда потоки имеют циклическую зависимость от пары синхронизированных объектов.**





Правила happens-before

1. **Освобождение мьютекса** *happens-before* **захват этого же монитора другим потоком.**
2. **Метод Thread.start()** *happens-before* **Thread.run().**
3. **Завершение метода run()** *happens-before* **выход из метода join().**
4. **Запись в volatile переменную** *happens-before* **чтение из той же переменной.**