Многопоточное программирование в Java

Владимир Поляков
vladimir.p.polyakov@gmail.com

https://github.com/drxaos-edu/lecture-java-concurrency

Зачем?

- Задействуют все процессоры
- Реализованы до появления многопроцессорных систем
  - Отзывчивый графический интерфейс
  - Нет простоя при ожидании ресурсов
  - Модель «поток на запрос»
  - Фоновые системные процессы

Многопоточность в Java

- Поддержка потоков на уровне языка
- Множество контекстов и счетчиков команд в едином пространстве памяти
- Со всеми объектами можно работать из любого потока
- Принципиальная неопределенность последовательности исполнения
- Но язык предоставляет средства, чтобы этой неопределенности избежать

Запуск потока

Как создать поток:

```
class MyThread extends Thread {
  public void run() { /* thread body */ }
}
```

```
MyThread mt = new MyThread(); // Create thread
mt.start();       // Starts thread running at run()
                  // Returns immediately
```

Thread

- Поток – это отдельный счетчик команд, а также стек, локальные переменные и т.д.
- Поток не является объектом
- Классы, объекты, методы и т.д. не принадлежат ни одному потоку
- Любой метод может быть выполнен любым потоком или несколькими потоками, причем даже одновременно

sleep()

С какой частотой выводится строка «Tick»?

```
public void run() {
  for(;;) {
    try {
      sleep(1000); // Pause for 1 second
    } catch (InterruptedException e) {
       return;       // caused by thread.interrupt()
    }
    System.out.println("Tick");
  }
}
```

Остановка потока

- Thread.stop()
- Thread.suspend()
- Thread.resume()

@Deprecated

- Thread.interrupt()
- Thread.interrupted()
- InterruptedException

volatile

(изменчивый, не постоянный)

- Для переменных, которые используются разными потоками
- Значение переменной, объявленной без volatile, может кэшироваться отдельно для каждого потока

Гонки (race condition)

- В многопоточном мире всегда считайте, что кто-то другой сейчас работает с вашими объектами
- Остальные потоки – ваши соперники

Гонки (race condition)

Вот что произойдет при одновременном чтении и записи:

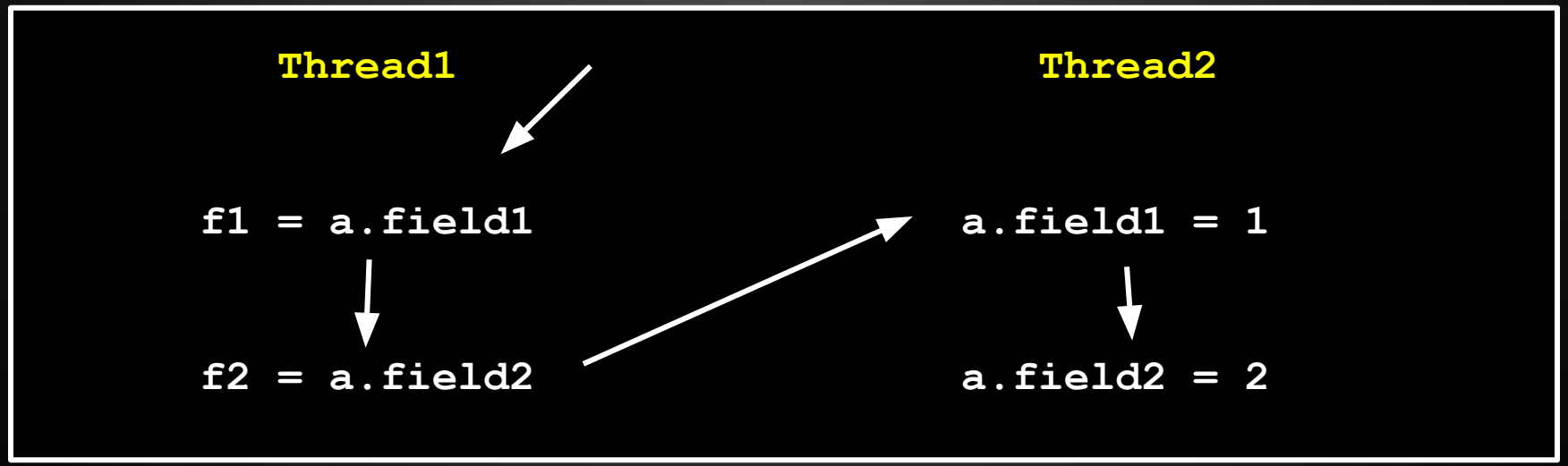
```
Thread1                 Thread2

f1 = a.field1           a.field1 = 1

f2 = a.field2           a.field2 = 2
```

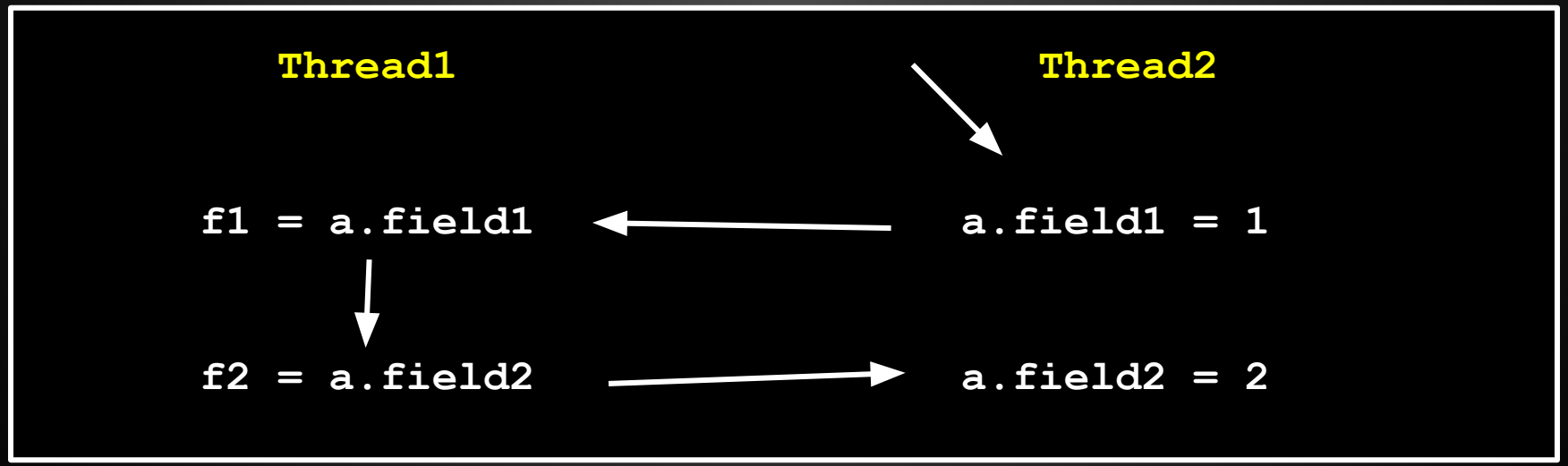
Гонки (race condition)

- Thread 1 выполняется первым
- Thread 1 получает исходные значения



Гонки (race condition)

- Попеременное выполнение
- Одно старое и одно новое значение



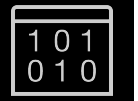
Неатомарные операции

- 32-битные чтение и запись гарантированно атомарны
- 64-битные операции могут быть неатомарными

Поэтому,

```
int i; double d;

Thread 1      Thread 2
i = 10;       i = 20;       В i будет 10 или 20
d = 10.0;     d = 20.0;     В d вероятно будет мусор
```



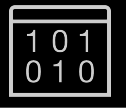
Блокировки в объектах (monitor)

- Каждый объект в Java имеет блокировку которую может получить как минимум один поток
- Поток останавливается и ждет, если он попытался получить блокировку, которую кто-то уже забрал
- Блокировка – это счетчик: один поток может заблокировать объект несколько раз

Оператор synchronized

- Оператор synchronized получает блокировку объекта перед тем как зайти в блок команд

```
Counter mycount = new Counter;
synchronized(mycount) { // получить блокировку mycount
  mycount.count();
}
```

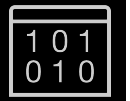


- Освобождает блокировку, когда блок команд заканчивается
- Объект для блокировки выбирает разработчик

Модификатор synchronized

```
class AtomicCounter {    // получить блокировку
  private int _count;    // объекта AtomicCounter
                         // перед выполнением метода

  public synchronized void count() {
    _count++;
  }
}
```



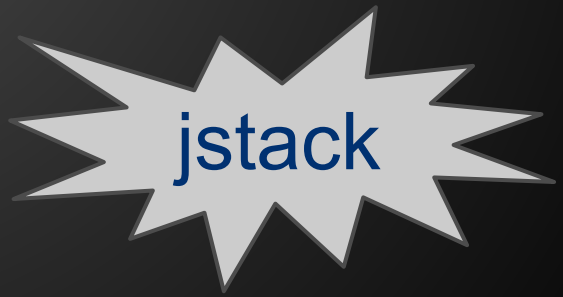
Такая реализация гарантирует, что только один поток сможет увеличивать счетчик в один момент времени

Deadlock

```
synchronized(Foo) {
  synchronized(Bar) {
    /* Deadlocked */
  }
}
```


```
synchronized(Bar) {
  synchronized(Foo) {
    /* Deadlocked */
  }
}
```

Правило: всегда получайте блокировки в одинаковом порядке



Приоритеты

- Каждый поток имеет приоритет от 1 до 10 (обычно 5)
- Задача планировщика – обеспечивать постоянное выполнение потоков с самым высоким приоритетом
- thread.setPriority(5)

Приоритеты

Из The Java Language Specification

Каждый поток имеет приоритет.

Если возникает конкуренция за вычислительные ресурсы, выполнение потоков с наивысшим приоритетом предпочитается выполнению потоков с низким приоритетом.

Однако, это вовсе не гарантия того, что потоки с наивысшим приоритетом будут непрерывно исполняться, **поэтому приоритеты потоков не могут быть использованы для надежной реализации взаимного исключения**.

Потоки с одним приоритетом

- Вызов yield() останавливает текущий поток, чтобы могли выполниться другие потоки с таким же приоритетом
- Реализация для Solaris исполняет потоки пока они сами не остановятся (wait(), yield(), etc.)
- Реализация для Windows использует квантование

Голодание

- Нечестный (упрощенный) планировщик
- Высокоприоритетные потоки могут потреблять все вычислительные ресурсы, не давая другим потокам выполняться
- Это называется голодание потоков (Starvation)

Ожидание условия

Допустим вы хотите, чтобы поток подождал выполнения какого-то условия, прежде чем продолжить выполнение

- Бесконечный цикл может вызвать deadlock

```
while (!condition) {}
```

- Вызовы yield() предотвращают deadlock, но очень неэффективно

```
while (!condition) yield();
```

wait() и notify()

- **wait()** похож на **yield()**, но требует, чтобы текущий поток потом кто-нибудь разбудил

```
while (!condition) wait();
```

- Поток, который может повлиять на проверяемое условие вызывает **notify()**, чтобы разбудить ожидающий поток
- Разработчик должен следить, чтобы каждому **wait()** соответствовал **notify()**

wait() и notify()

У каждого объекта есть набор потоков, ожидающих освобождения его блокировки (wait set)

```
synchronized (obj) {   // Получить блокировку obj
  obj.wait();          // остановиться

   // добавление потока в
   // wait set объекта
   // освобождает блокировки obj

В другом потоке:
obj.notify();       // разбудить ждущий поток
```

wait() и notify()

1. Thread 1 получает блокировку объекта
2. Thread 1 вызывает wait() на объекте
3. Thread 1 освобождает блокировку, добавляет себя в wait set объекта

4. Thread 2 вызывает notify() на объекта (должен иметь блокировку объекта для вызова notify())

5. Thread 1 разбужен: удаляется из wait set
6. Thread 1 получает блокировку
7. Thread 1 продолжает выполнение после wait()

wait() и notify()

- **notify()** случайно выбирает и будит один поток (из многих)

- **notifyAll()** будит все ждущие потоки

Пишем Blocking Buffer

- Буфер для записи и чтения
- Поток чтения блокируется если нет данных для чтения
- Поток записи блокируется если данные еще не прочитаны

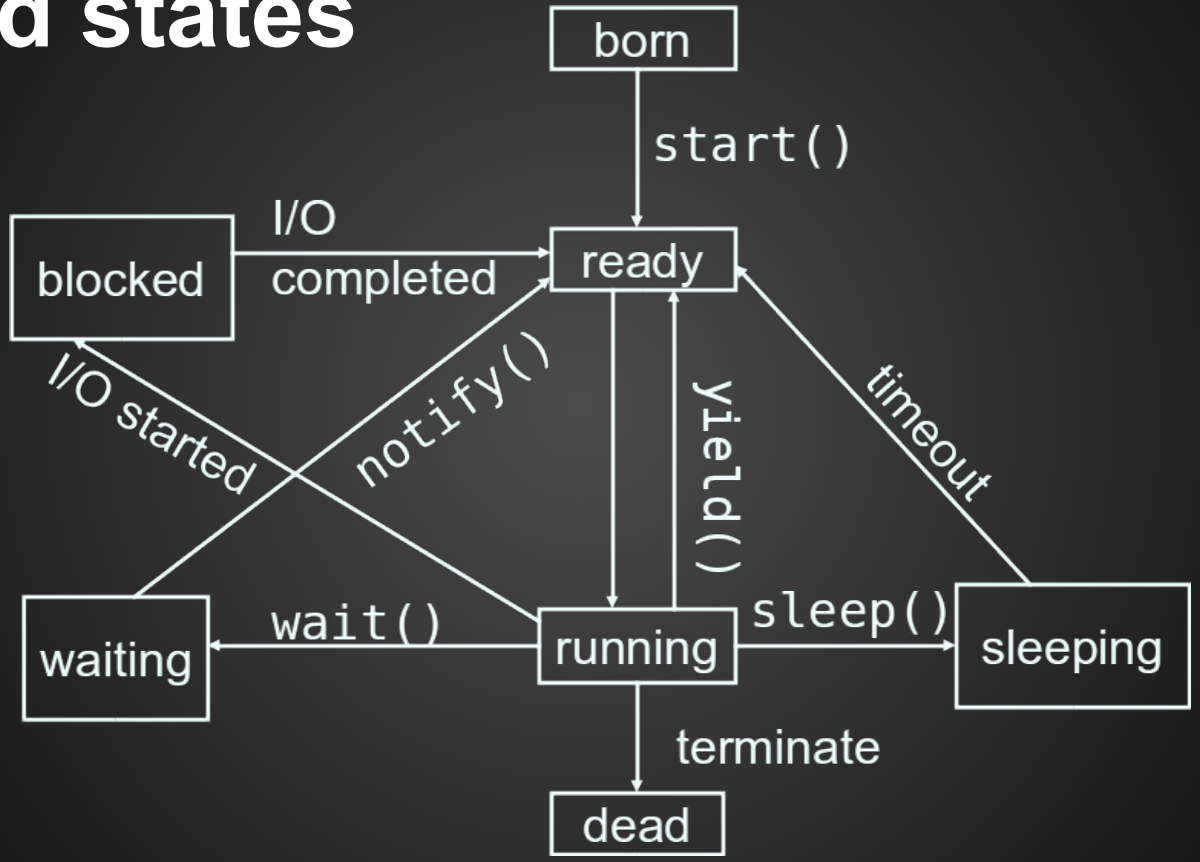
```
class OnePlace {
  El value;

  public synchronized void write(El e) { … }
  public synchronized El read() { … }
}
```

Пишем Blocking Buffer

```
synchronized void write(El e) throws InterruptedException
{
  while (value != null) wait();   // Block while full
  value = e;
  notifyAll();                    // Awaken any waiting read
}
public synchronized El read() throws InterruptedException
{
  while (value == null) wait();   // Block while empty
  El e = value; value = null;
  notifyAll();                    // Awaken any waiting write
  return e;
}
```

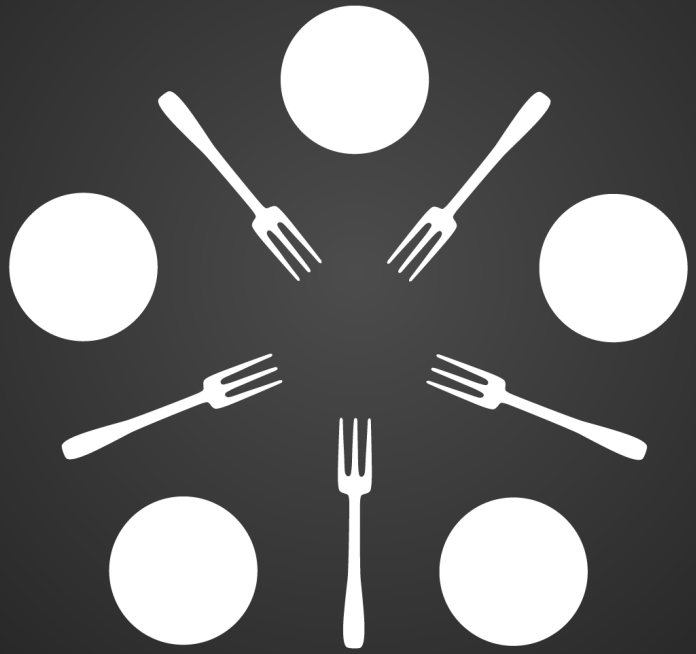
Thread states



Счастливый билет

http://ru.wikipedia.org/wiki/Счастливый_билет

Обедающие философы



Вопросы

vladimir.p.polyakov@gmail.com
https://github.com/drxaos-edu