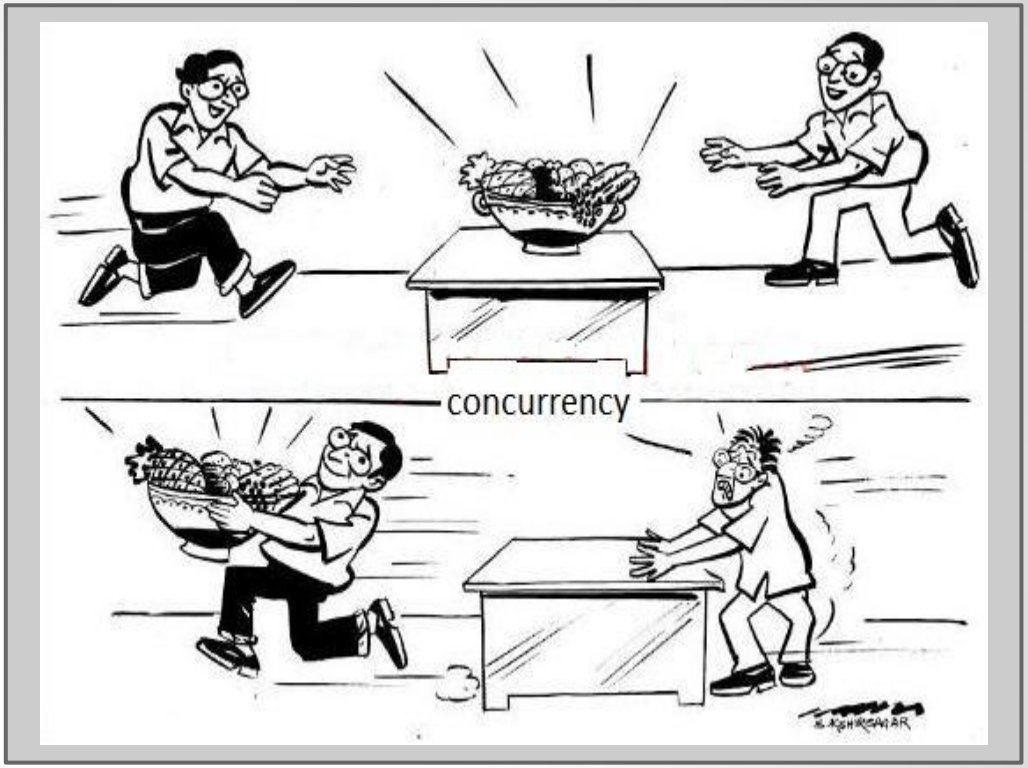
Потоки в Java

План



- Зачем?
- Что?
- Как?
- Почему не работает?

Зачем?

- утилизация ресурсов CPU
- абстракция многозадачности
- асинхронность
- производительность

Что

```
class MyThread extends Thread {

  @Override
  public void run() {
    …
  }
}


Thread t = new MyThread();
t.start();
```

```
class MyTask implements Runnable {

  @Override
  public void run() {
    …
  }
}


Thread t = new Thread(new MyTask());
t.start();
```

Timer

```
Timer timer = new Timer();


timer.schedule(new TimerTask() {
  @Overrid
  public void run() {
    ...
  }}, 2*60*1000);
```

start() / run()

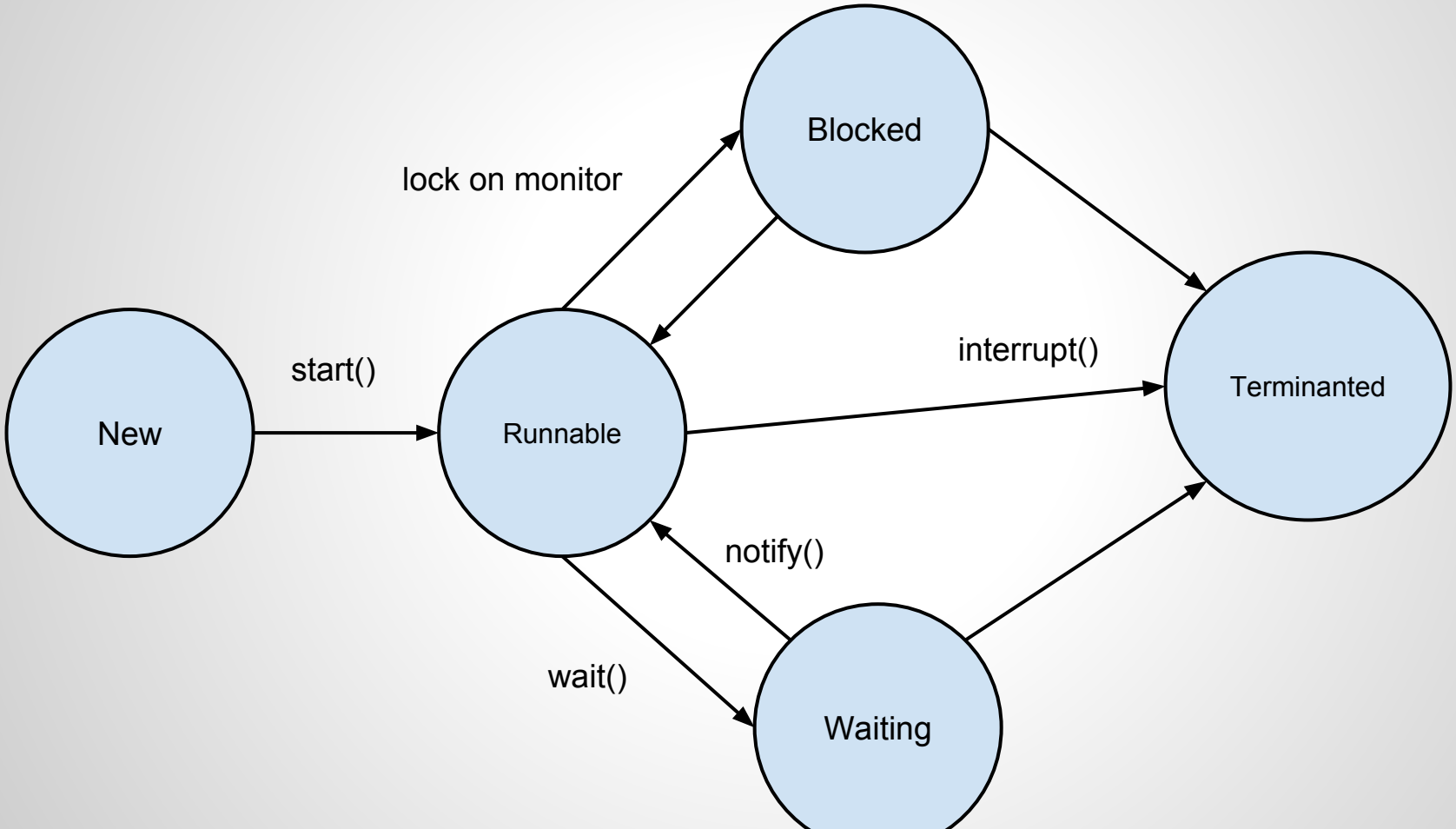
Запустить задачу в отдельном потоке

thread.start()

код, который будет выполняться

```
interface Runnable {
    run() {...}
}
```

*Если вызвать Thread.run() - то код выполнится в текущем потоке!


Blocked
lock on monitor
New
start()
Runnable
interrupt()
Terminanted
notify()
wait()
Waiting

Нельзя дважды запустить один и тот же поток!

```
Thread t = new MyThread();

t.start();

t.start(); // ошибка
```

- **NEW -** поток был создан, но еще не запущен (start())**;**
- **RUNNABLE -** выполняющийся JVM;
- **BLOCKED -** заблокированный на мониторе;
- **WAITING -** ждет действий от другого потока;
- **TERMINATED -** исполнение завершено;

- start()
- join() - дождаться выполнения потока
- static sleep() - уснуть
- yield() - отдать управление

А как остановить?

- thread.interrupt() - выставить флаг
- thread.isInterrupted() - проверить флаг
- * static Thread.interrupted() - проверить флаг текущего потока

- Внутри потока в цикле проверять флаг isInterrupted()
- На блокирующих операциях обрабатывать InterruptedException() и корректно завершать работу потока

Если не написать код обработки прерывания, поток не завершится!

ExecutorService

```
ExecutorService service = Executors.newFixedThreadPool(10);
service.submit(new Runnable(...));
```

Пул потоков - автоматически создает и запускает потоки

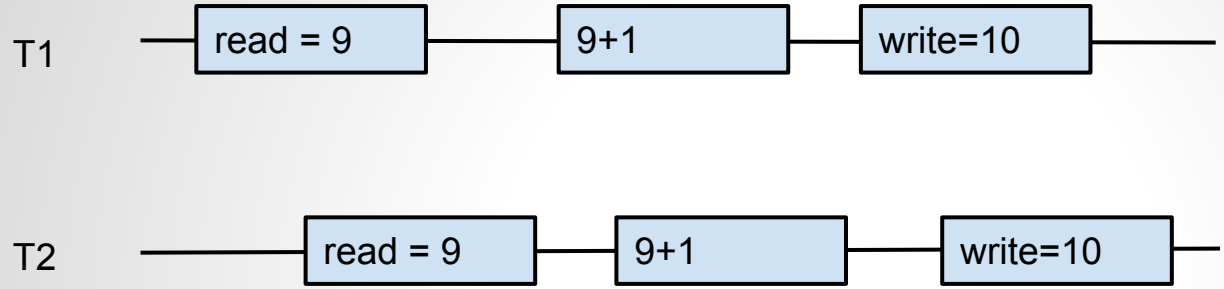
- Ограниченное число потоков (fixed)
- Создается сколько нужно (cached)

Учимся считать

```
class UnsafeSequence {
    private int val;

    public int nextVal() {
        return val++;
    }
}
```

Учимся считать

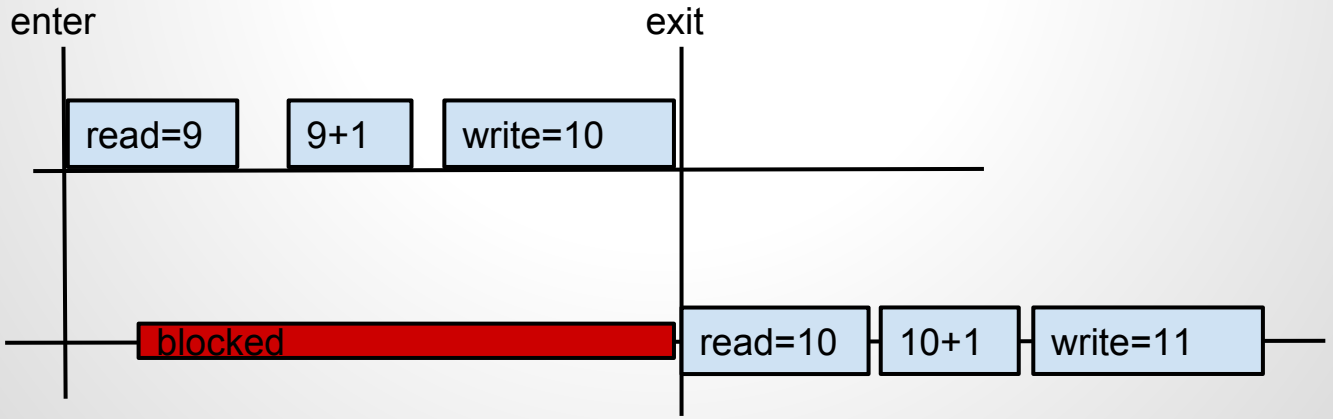


- считать значение
- изменить значение
- записать значение

Чиним

```
class SafeSequence {
    private int val;
    public synchronized int nextVal() {return val++;}
}
```

synchronized - критическая секция, эксклюзивный доступ

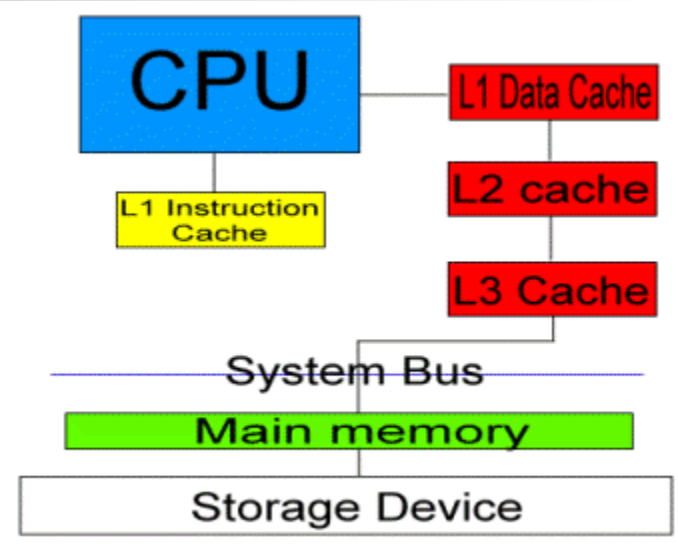


Атомарность

Атомарные операции

- чтение и запись примитивных типов (кроме long, double)
- чтение и запись ссылок
- volatile гарантирует атомарность для long/double

Устройство CPU



- Основная память - медленно
- Хранит данные в кэше
- Синхронизирует кэш
- Процессоров может быть много и у каждого свой кэш
- Синхронизация между процессорами

Видимость

При каких условиях один поток видит изменения, сделанные другим потоком

volatile - гарантирует видимость чтения/записи переменной между потоками

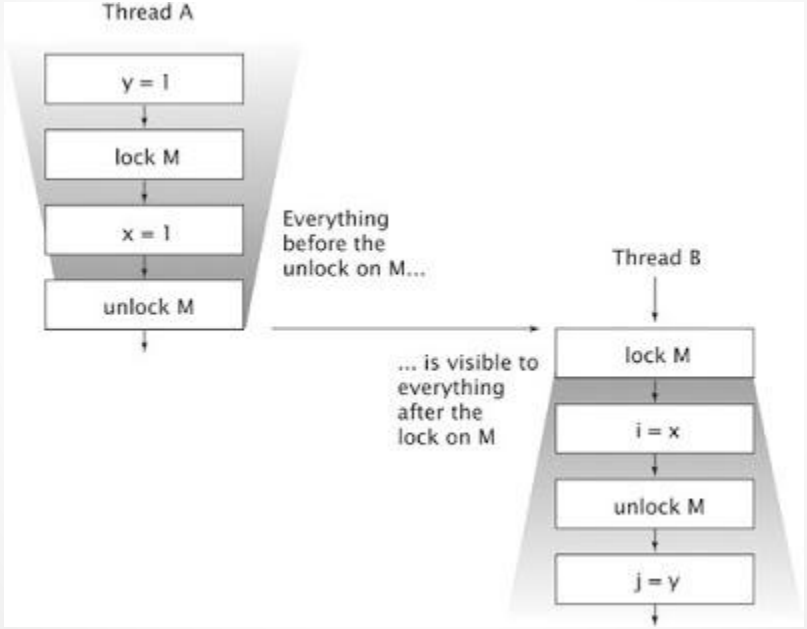
Reordering

С точки зрения наблюдающего (другого потока) инструкции могут идти в другом порядке, чем написано в коде (program order)

```
//first thread
result = calc();
isReady = true;
```

```
//second thread
if (isReady){
    takeDesision(result);
}
```

Happens-Before



synchronized / lock

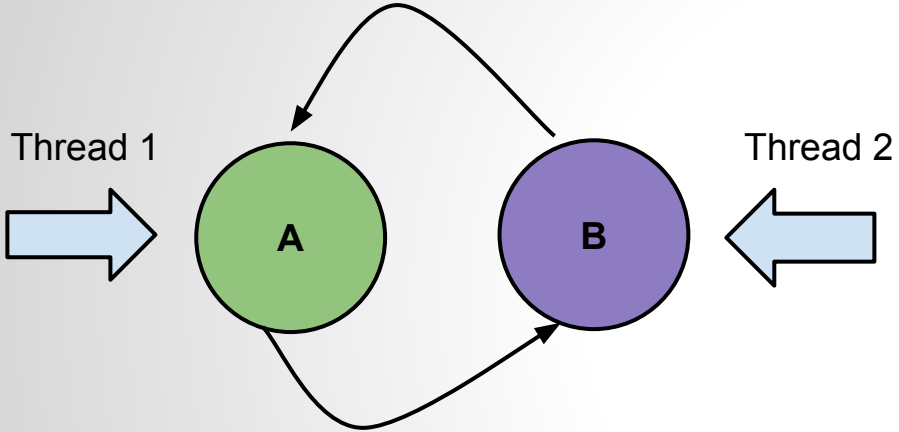
- метод
- блок
- статический метод - на объекте class

- С каждым java-объектом связан монитор
- Монитор позволяет разграничить доступ
- У монитора может быть только один владелец

synchronized / lock

- потоки борются за владение монитором
- fair lock - захватывает, который ждет дольше всех (`ReentrantLock(boolean fair)`)
- lock contention - из-за нагрузки на критическую секцию, потоки простаивают

Deadlock



1. T1 acquire A
2. T2 acquire B
3. T1 waiting for B
4. T2 waiting for A
5. deadlock :(

Deadlock

- Упорядочить взятие блокировок
- Использовать tryLock()
- Использовать таймаут

wait() / notify()

- методы Object
- с каждым объектом связан монитор
- вызывать внутри критической секции (на этом же мониторе)
- wait() отпускает монитор
- notify()/notifyAll() не имеет порядка (очереди)

```
while (!isReady) {
   wait();
}
```

isReady - проверка логического условия

wait() - ожидание сигнала от другого потока

```
isReady = true;
notifyAll();
```

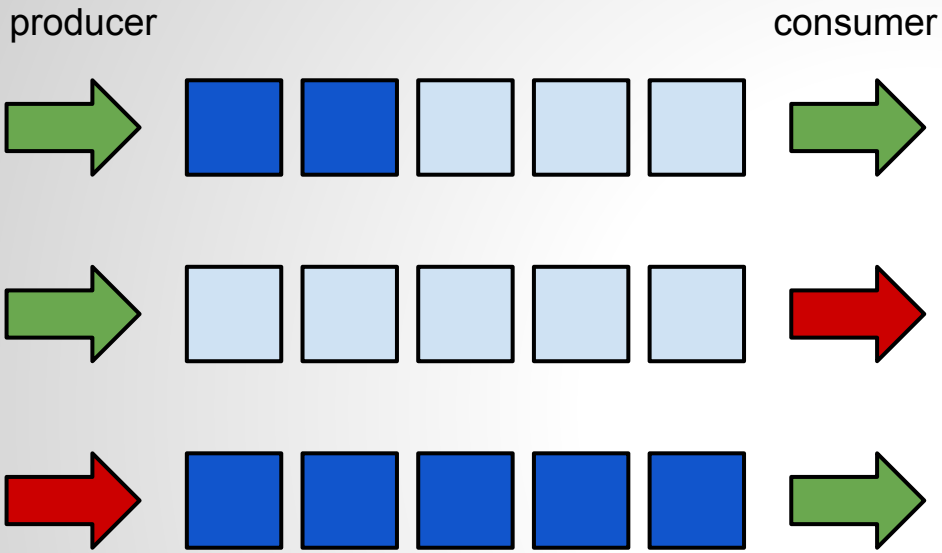
Producer / Consumer

Producer - источник данных

- данные готовы

Consumer - потребитель данных

- нет места



состояния:

- рабочее состояние
- очередь пуста (нечего брать)
- очередь полна (некуда класть)

java.util.concurrent