Самарский государственный аэрокосмический университет
имени академика С.П. Королёва

Объектно-ориентированное программирование

Многопоточное программирование: общие принципы и реализация в Java

Занятие 11

Самара
2015

План лекции

- Многопоточное программирование и его особенности
- Потоки и работа с ними
- Группы потоков
- Приоритеты потоков
- Демон-потоки
- Блокировки и синхронизация
- Новые виды ошибок
- Совместная работа с полями и переменными
- Методы класса Object
- Прерывание потоков
- Высокоуровневые средства

Проблемы однопоточного подхода

- Монопольный захват задачей процессорного времени
- Смешение логически несвязанных фрагментов кода
- Попытка их разделения приводит к возникновению в программе новых систем и усложнению кода

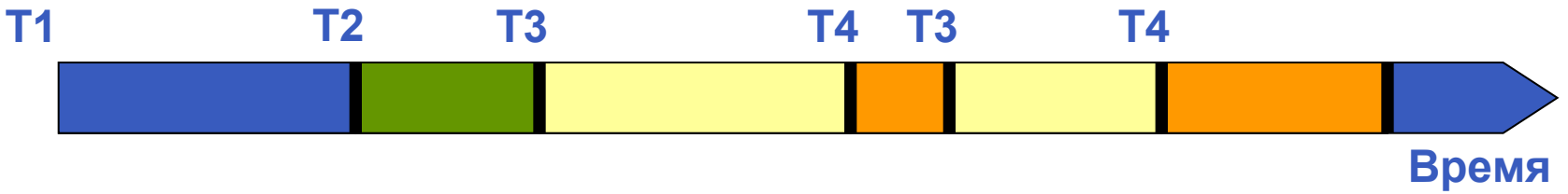
Многопоточное программирование

- Последовательно выполняющиеся инструкции составляют поток
- Потоки выполняются условно независимо
- Потоки могут взаимодействовать друг с другом
- В многоядерной системе поток монопольно занимает одно ядро

Это не так

Квантование времени (Time-Slicing)

- Время разделяется на интервалы (кванты времени)
- Во время одного кванта обрабатывается один поток команд
- Решение о выборе потока принимается до начала интервала
- Переключения между потоками с высокой частотой



- Иллюзия одновременности!

Особенности многопоточности

- Простота выделения подзадач
- Более гибкое управление выполнением задач
- Более медленное выполнение?
- Выигрыш в скорости выполнения при разделении задач по используемым ресурсам
- Выигрыш в скорости выполнения на многоядерных системах
- Недетерминизм при выполнении

Использование класса Thread

- Описание класса

```
public class ИмяКласса extends Thread {
  public void run() {
    // Действия, выполняемые потоком
  }
}
```

- Запуск потока

```
ИмяКласса t = new ИмяКласса();
t.start(); // именно start(), а не run() !!!
```

Использование интерфейса Runnable

- Описание класса

```
public class ИмяКласса implements Runnable {
  public void run() {
    // Действия, выполняемые потоком
  }
}
```

- Запуск потока

```
Runnable r = new ИмяКласса(); // Это ещё не поток
Thread t = new Thread(r); // А вот это уже поток
t.start();
```

Особенности использования интерфейса Runnable

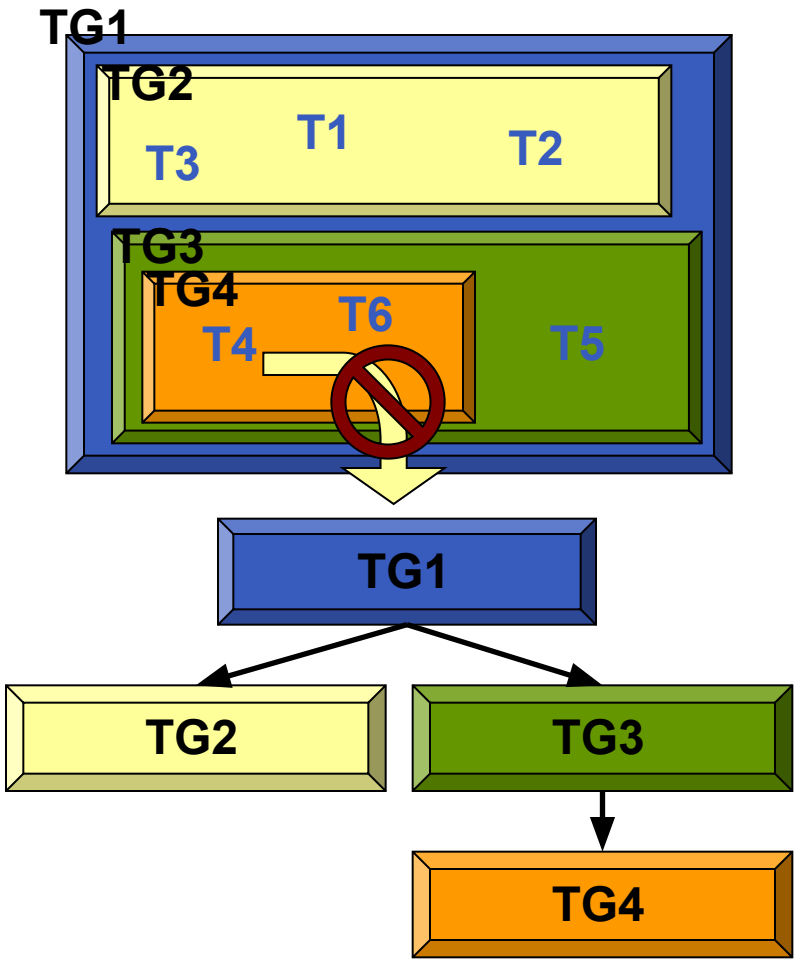
- Возможность создать класс, описывающий тело потока и наследующий от класса, отличного от `Thread`
- Объект вашего класса не является объектом потока
- Невозможно использовать напрямую методы класса `Thread`
- Можно получить ссылку на объект текущего потока с помощью статического метода `currentThread()` класса `Thread`

Управление потоками

- `void start()`
  Запускает выполнение потока
- `void stop()`
  Прекращает выполнение потока
- `void suspend()`
  Приостанавливает выполнение потока
- `void resume()`
  Возобновляет выполнение потока
- `void join()`
  Останавливает выполнение текущего потока до завершения потока, у объекта которого был вызван метод
- `static void sleep(long millis)`
  Останавливает выполнение текущего потока как минимум на `millis` миллисекунд
- `static void yield()`
  Приостанавливает выполнение текущего потока, предоставляет возможность выполнять другие потоки

Группы потоков (ThreadGroup)

- Каждый поток находится в группе
- Группы потоков образуют дерево, корнем служит начальная группа
- Поток не имеет доступа к информации о родительской группе
- Изменение параметров и состояния группы влияет на все входящие в нее потоки



Создание групп потоков

- Создание группы

```
//Без явного указания родительской группы
ThreadGroup group1 = new ThreadGroup("Group1");
//С явным указанием родительской группы
ThreadGroup group2 = new ThreadGroup(group1, "Group2");
```

- Создание потока

```
//Без явного указания группы
MyThread t1 = new MyThread("Thread1");
//С явным указанием группы
MyThread t2 = new MyThread(group2, "Thread2");
```

Операции в группе потоков

- `int activeCount()`
  Возвращает оценку количества потоков
- `int enumerate(Thread[] list)`
  Копирует в массив активные потоки
- `int activeGroupCount()`
  Возвращает оценку количества подгрупп
- `int enumerate(ThreadGroup[] list)`
  Копирует в массив активные подгруппы
- `void interrupt()`
  Прерывает выполнение всех потоков в группе

Приоритеты потоков

- Приоритет – количественный показатель важности потока
- Недетерминированно воздействуют на системную политику упорядочивания потоков
- Базовый алгоритм программы не должен зависеть от схемы расстановки приоритетов потоков
- При задании значений приоритетов рекомендуется использовать константы

Приоритеты потоков

- Константы в классе **Thread**
  ```
  MAX_PRIORITY
  MIN_PRIORITY
  NORM_PRIORITY
  ```
- Методы потока
  ```
  int getPriority()
  void setPriority(int newPriority)
  ```
- Методы группы потоков
  ```
  int getMaxPriority()
  void setMaxPriority(int priority)
  ```

Демон-потоки (Daemons)

- Демон-потоки позволяют описывать фоновые процессы, которые нужны только для обслуживания основных потоков выполнения и не могут существовать без них
- Уничтожаются виртуальной машиной, если в группе не осталось не-демон потоков
- `void setDaemon(boolean on)`
  Устанавливает вид потока
  Вызывается до запуска потока
- `boolean isDaemon()`
  Возвращает вид потока:
  `true` – демон, `false` – обычный

Демон-группы потоков

- Демон-группа автоматически уничтожается при остановке последнего ее потока или уничтожении последней подгруппы потоков
- `void setDaemon(boolean on)`
  Устанавливает вид группы
- `boolean isDaemon()`
  Возвращает вид группы:
  `true` – демон, `false` – обычная

Неконтролируемое совместное использование ресурсов

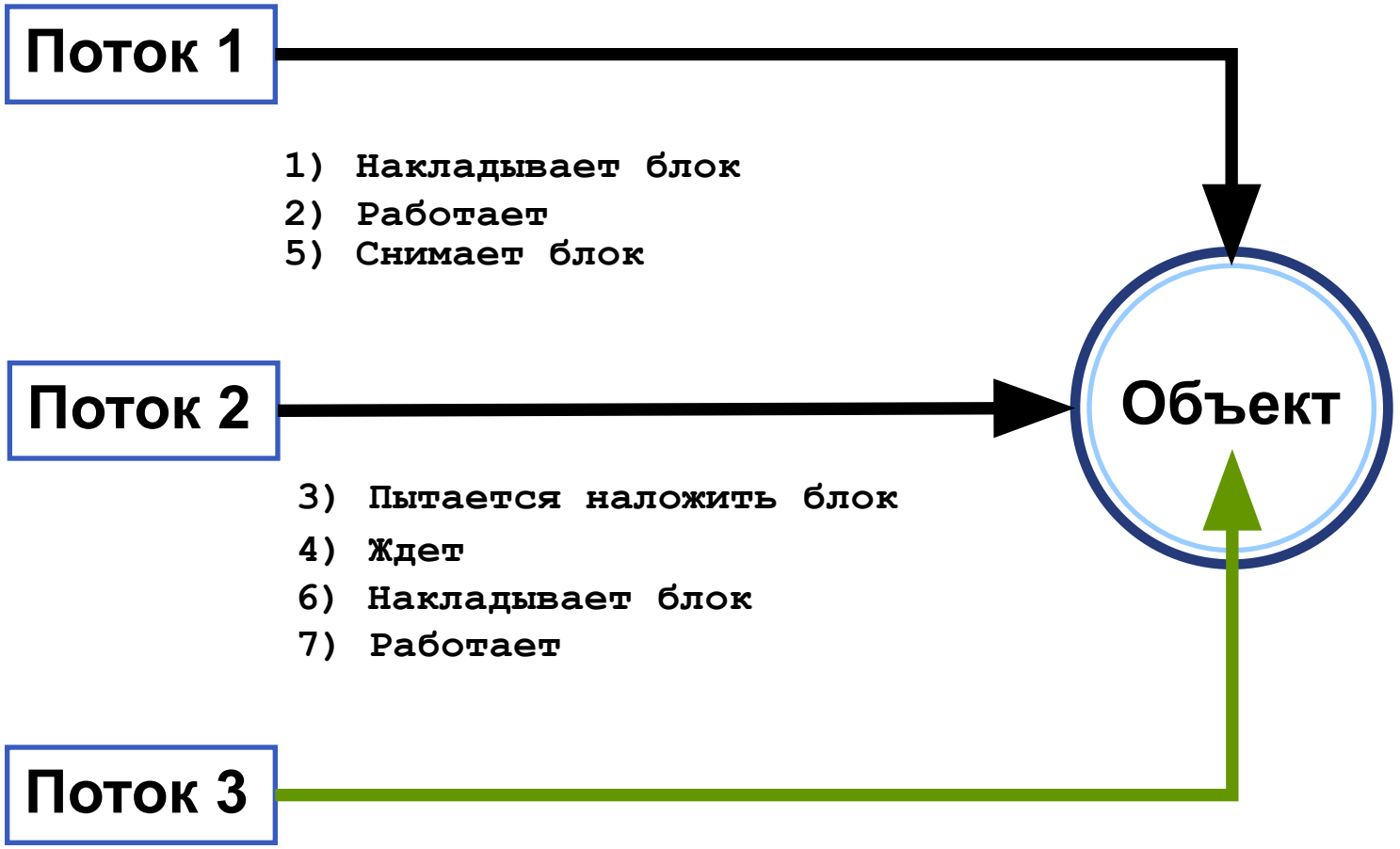
- Недетерминизм программы
  Конечный результат работы программы непредсказуем

- Некорректность работы программы
  Возможность некорректной работы алгоритма, возникновения исключительных ситуаций

Блокировки

- Только один поток в один момент времени может установить блокировку на некоторый объект
- Попытка блокировки уже заблокированного объекта приводит к останову потока до момента разблокирования этого объекта
- Наличие блокировки не запрещает всех остальных действий с объектом

Блокировки


Поток 1
1) Накладывает блок
2) Работает
5) Снимает блок
Поток 2
Объект
3) Пытается наложить блок
4) Ждет
6) Накладывает блок
7) Работает
Поток 3

Синхронизация

- Синхронизированный блок

```
//Блокируется указанный объект
synchronized (ссылкаНаОбъект>) {
    // Тело блока синхронизации
}
```

- Синхронизированный метод

```
//Блокируется объект-владелец метода
public synchronized void метод() {
    // Тело метода
}
```

Новые виды ошибок

- Отсутствие синхронизации
- Необоснованная длительная блокировка объектов
- Взаимная блокировка (deadlock)
- Возникновение монопольных потоков
- Нерациональное назначение приоритетов

Совместная работа с полями и переменными

- Значения переменных изменяются атомарным образом (кроме `double` и `long`)
- При совместной работе с полем может возникнуть неоднозначность
  - Например, на объект, на который ссылается переменная, наложена блокировка, после чего значение переменной изменяется
  - Например, компилятор может оптимизировать фрагмент кода, предполагая, что поле не изменяет значение

Модификатор полей и переменных final

- После первого присвоения переменная не может изменять своё значение
- Если блокировка накладывается на объект, ссылка на который хранится в поле, поле обычно делают неизменяемым
- Локальные и анонимные классы могут обращаться к локальным переменным, только если они неизменяемы

Модификатор полей volatile

- Предупреждает компилятор о том, что переменная может изменить своё значение в произвольный момент времени
- Обращение к переменной всегда будет возвращать именно последнее присвоенное ей значение
- Если работа с полем ведётся только в синхронизированном коде, применение модификатора неосмысленно

Специальные методы класса Object

- Каждый объект имеет набор ожидающих потоков исполнения (wait-set)
- Любой поток может вызвать метод `wait()` любого объекта и попасть в его wait-set, остановившись до пробуждения
- Метод объекта `notify()` пробуждает один, случайно выбранный поток из wait-set объекта
- Метод объекта `notifyAll()` пробуждает все потоки из wait-set объекта

Особенности использования методов класса Object

- Метод может быть вызван потоком у объекта только после установления блокировки на этот объект
- Потоки, прежде чем приостановить выполнение после вызова метода `wait()`, снимают все свои блокировки
- После вызова освобождающего метода потоки пытаются восстановить ранее снятые блокировки

Запрещенные действия над потоками

- `Thread.suspend()`, `Thread.resume()`

  Увеличивает количество взаимных блокировок

- `Thread.stop()`

  Использование приводит к возникновению поврежденных объектов

Корректное прерывание потока

- `public void interrupt()`
  Изменяет статус потока на прерванный
- `public static boolean interrupted()`
  Возвращает и очищает статус потока (прерван или нет)
- `public boolean isInterrupted()`
  Возвращает статус потока (прерван или нет)
- Поток должен в ходе своей работы проверять свой статус и корректно завершать работу, если его прервали

А если поток «спит»?

- В том случае, если в текущий момент поток выполняет методы `wait()`, `sleep()`, `join()`, а его прерывают вызовом метода `interrupt()`…
- метод прерывает свое выполнение с выбросом исключения `InterruptedException`!
- Потоку не сообщается, что его прервали!

Пример простого семафора

```
public class Semaphore {
    private boolean canWrite = true;

    public synchronized void beginRead()
        throws InterruptedException {
        while (canWrite) {
            wait();
        }
    }

    public synchronized void endRead() {
        canWrite = true;
        notifyAll();
    }
```

Пример простого семафора

```
    public synchronized void beginWrite()
        throws InterruptedException {
        while (!canWrite) {
            wait();
        }
    }

    public synchronized void endWrite() {
        canWrite = false;
        notifyAll();
    }
}
```

java.util.concurrent

- Пакет содержит высокоуровневый инструментарий для многопоточных приложений
- Пакет содержит следующие категории инструментов
  - Executors – средства запуска потоков
  - Synchronizers – средства синхронизации работы потоков
  - Timing – вспомогательные средства контроля времени
  - Concurrent structures – структуры, корректно работающие в многопоточных приложениях (без блокировки всей структуры)

java.util.concurrent

- `java.util.concurrent.atomic` пакет содержит классы оберток для базовых типов, обеспечивающие корректный доступ к значениям в многопоточных приложениях
- `java.util.concurrent.locks` пакет содержит высокоуровневые средства работы с блокировками и критическими секциями

Спасибо за внимание!

Дополнительные источники

- Арнолд, К. Язык программирования Java [Текст] / Кен Арнолд, Джеймс Гослинг, Дэвид Холмс. – М. : Издательский дом «Вильямс», 2001. – 624 с.
- Вязовик, Н.А. Программирование на Java. Курс лекций [Текст] / Н.А. Вязовик. – М. : Интернет-университет информационных технологий, 2003. – 592 с.
- Хорстманн, К. Java 2. Библиотека профессионала. Том 2. Тонкости программирования [Текст] / Кей Хорстманн, Гари Корнелл. – М. : Издательский дом «Вильямс», 2010 г. – 992 с.
- Эккель, Б. Философия Java [Текст] / Брюс Эккель. – СПб. : Питер, 2011. – 640 с.
- JavaSE at a Glance [Электронный ресурс]. – Режим доступа: http://www.oracle.com/technetwork/java/javase/overview/index.html, дата доступа: 21.10.2011.
- JavaSE APIs & Documentation [Электронный ресурс]. – Режим доступа: http://www.oracle.com/technetwork/java/javase/documentation/api-jsp-136079.html, дата доступа: 21.10.2011.