Многопоточное программирование

Лекция 7

Самара
2020

NetCracker®

План лекции

- Потоки инструкций и многопоточное программирование
- Создание потоков и управление ими
- Совместное использование ресурсов и блокировки
- Взаимодействие между потоками

Проблемы однопоточного подхода

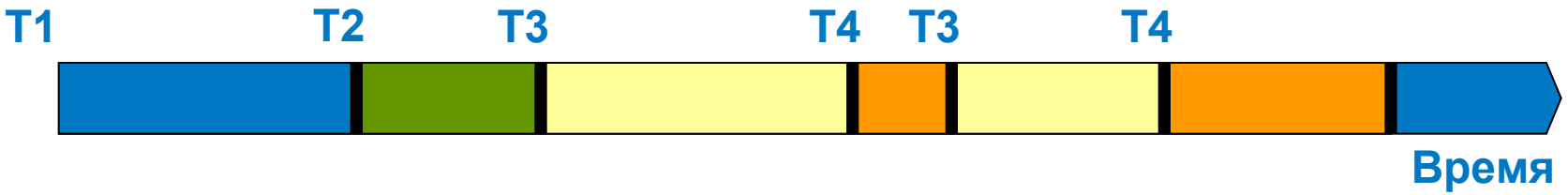
- Монопольный захват задачей процессорного времени
- Смешение логически несвязанных фрагментов кода
- Попытка их разделения приводит к возникновению в программе новых систем и усложнению кода

Многопоточное программирование

- Последовательно выполняющиеся инструкции составляют поток
- Потоки выполняются условно независимо
- Потоки могут взаимодействовать друг с другом

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

Класс Thread

- Поток выполнения представляется экземпляром класса `Thread`
- Для создания потока выполнения можно
  - создать класс, наследующий `Thread`
  - переопределить метод `run()`
- Для запуска потока используется метод `start()` у объекта класса-наследника

Использование класса Thread

- Описание класса

```
public class <Имя класса> extends Thread {
  public void run() {
    // Действия, выполняемые потоком
  }
}
```

- Запуск потока

```
<Имя класса> t = new <Имя класса>();
t.start(); //именно start(), а не run() !!!
```

Интерфейс Runnable

- Объявляет один метод – `void run()`
- Объект данного типа не является потоком
- Невозможно использовать напрямую методы класса `Thread`
- Возможность создать класс, описывающий тело потока и наследующий от класса, отличного от `Thread`
- Можно получить ссылку на объект текущего потока с помощью статического метода `currentThread()` класса `Thread`

Использование интерфейса Runnable

- Описание класса

```
public class <Имя класса> implements Runnable {
  public void run() {
    // Действия, выполняемые потоком
  }
}
```

- Запуск потока

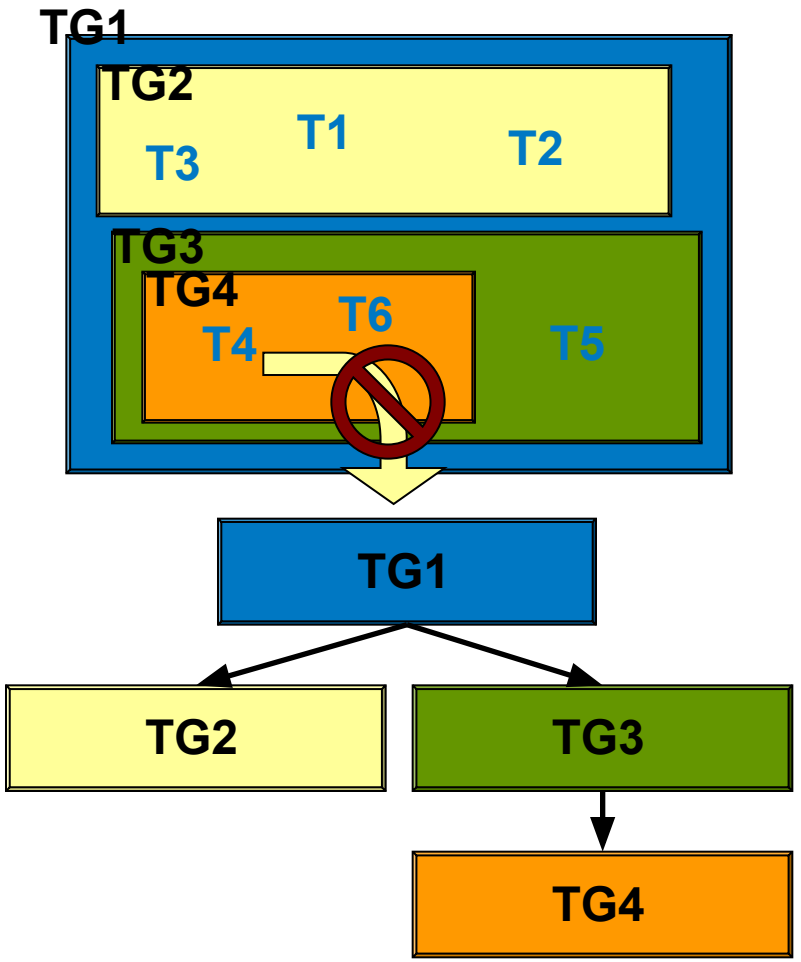
```
Runnable r = new <Имя класса>();
Thread t = new Thread(r);
t.start();
```

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 t = new MyThread("Thread1");
//С явным указанием группы
MyThread t = 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
```
static int MAX_PRIORITY
static int MIN_PRIORITY
static int NORM_PRIORITY
```
- Методы потока
```
int getPriority()
void setPriority(int newPriority)
```
- Методы группы потоков
```
int getMaxPriority()
void setMaxPriority(int priority)
```

Приоритеты потоков

```
public class MyThread extends Thread {
 public void run() {
  long sum = 0;
  for (int i = 0; i < 1000000; i++) {
   if (i % 1000 == 0) {System.out.println(getName() + ": " +
i/1000);}
  }
 }
}
```

```
Thread t1 = new MyThread();
t1.setPriority(Thread.MIN_PRIORITY);
t1.start();

Thread t2 = new MyThread();
t2.setPriority(Thread.MAX_PRIORITY);
t2.start();
```

Приоритеты потоков

```
Thread-0: 0
Thread-1: 0
Thread-1: 1
Thread-0: 1
Thread-1: 2
Thread-0: 2
Thread-1: 3
Thread-0: 3
Thread-1: 4
Thread-0: 4
Thread-1: 5
Thread-0: 5
Thread-1: 6
Thread-0: 6
Thread-1: 7
```

```
Thread-0: 0
Thread-0: 1
Thread-1: 0
Thread-0: 2
Thread-1: 1
Thread-0: 3
Thread-1: 2
Thread-0: 4
Thread-1: 3
Thread-0: 5
Thread-1: 4
Thread-1: 5
Thread-1: 6
Thread-1: 7
Thread-1: 8
```

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

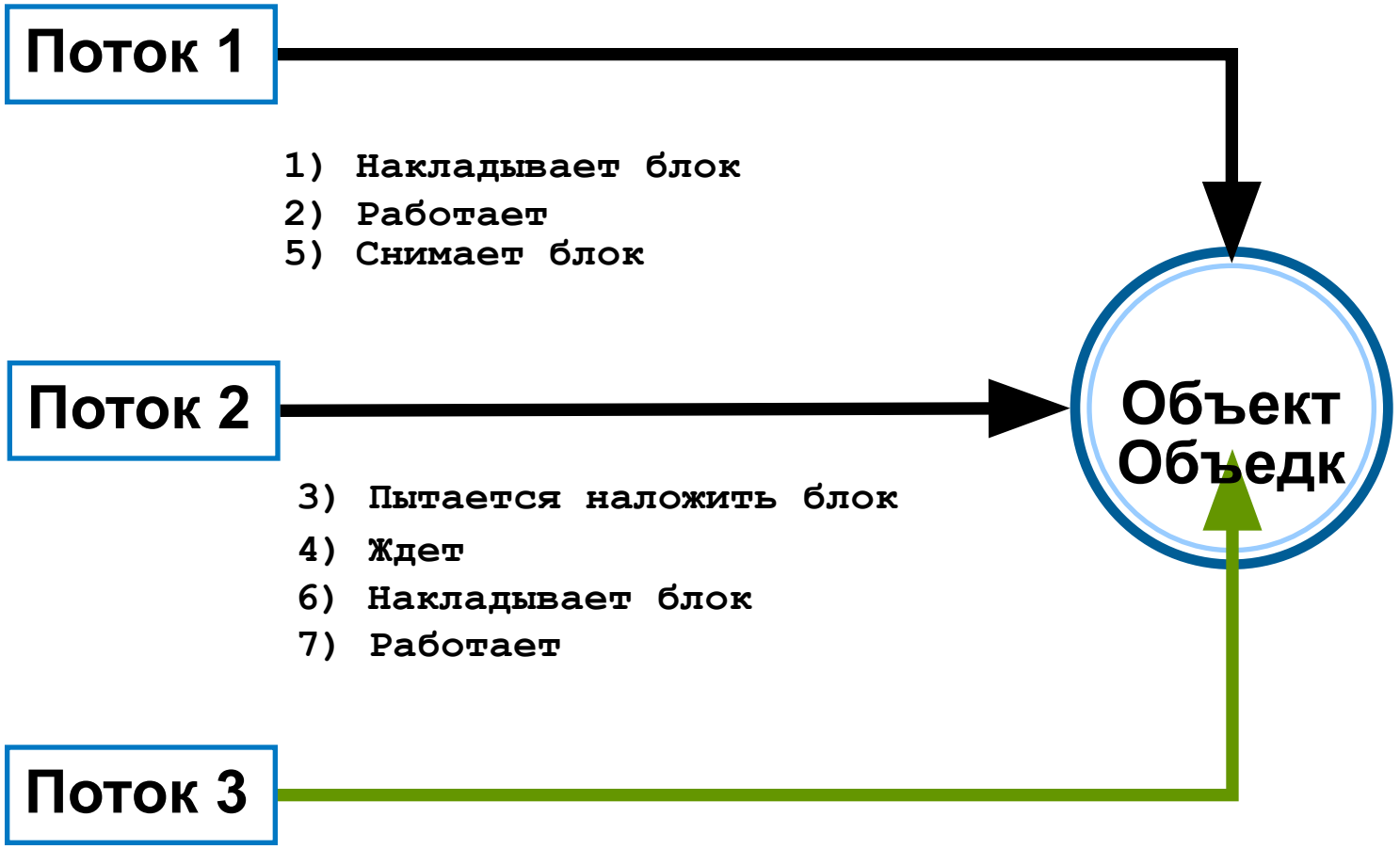
- Недетерминизм программы
  Конечный результат работы программы непредсказуем

- 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 остановке 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
3) Пытается наложить блок
4) Ждет
6) Накладывает блок
7) Работает
Поток 3
Объект
Объедк

Синхронизация

- Синхронизированный блок

```
//Блокируется указанный объект
synchronized (<Ссылка на объект>) {
    <Тело блока синхронизации>
}
```

- Синхронизированный метод

```
//Блокируется объект-владелец метода
public synchronized void <Имя метода>() {
    <Тело метода>
}
```

Характерные ошибки

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

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

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 кода объекта-посредника

```
public class Keeper {
    private Object data;
    private boolean newed = false;

    synchronized public void putData(Object obj)
        try {
            while(newed)
                wait();
            data = obj;
            newed = true;
            notifyAll();
        } catch(InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
...
```

Пример кода объекта-посредника

```
...

    synchronized public Object getData()
      throws InterruptedException {
        while(!newed)
            wait();
        newed = false;
        notifyAll();
        return data;
    }
}
```