Лекция 8. Потоки выполнения

NetCracker®

Введени
е

Проблемы однопоточного подхода:

- Монопольный захват задачей процессорного времени.
- Простой во время ожидания освобождения внешних
- ресурсов.

  Смешение логически несвязанных фрагментов, которые должны выполнятся параллельно.

Многопоточное программирование:

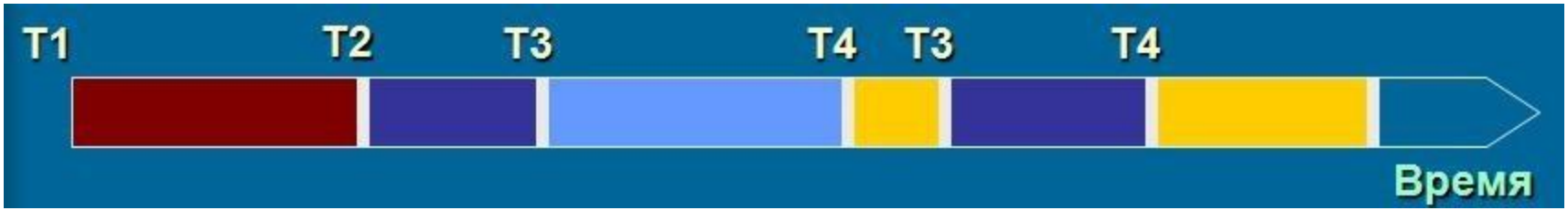
- Последовательно выполняющиеся инструкции составляют поток.
- Потоки выполняются условно независимо. Взаимодействие потоков друг с другом.

Квантование времени

Особенности процедуры квантования времени:

- Время делиться на интервалы (кванты времени)
- Во время одного кванта обрабатывается один поток данных
- Решение о выборе потока принимается до начала интервала
- Переключение между потоками с высокой частотой эмулирует многопоточность
- Поддержка приоритетов для выполняемых потоков

Иллюзия одновременности!



Особенности многопоточности

Особенности многопоточной архитектуры:

- Простота выделения подзадач
- Более гибкое управление выполнением задач
- Выигрыш в скорости выполнения при разделении задач по
- Более медленное выполнение задач
- Недетерминированное выполнение
- Использование ресурсами

Создание потоков. Класс Thread

Поток – экземпляр класса **Thread**. Для создания своего потока исполнения необходимо

наследоваться от данного класса и переопределить метод **run()**.

Пример класса:

```
public class MyThread extends Thread {
    public void run() {
        // некоторое долгое вычисление
        long sum = 0;
        for (int i=0; i < 1000; i++) {
            sum+=i;
        }
        System.out.println(sum);
    }
}
```

Способ запуска:

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

- **`run()`** – метод, который содержит

  действия, которые должны выполнятся в

- потоке.

  `start()` – унаследованный метод,

  который сообщает виртуальной машине,

  что необходимо запустить новый поток

  исполнения и начать в нём выполнение

Интерфейс Runnable

- Множественное наследование от класса **Thread** – может привести к конфликту!
- **Runnable** – интерфейс, в котором абстрагируется концепция некой сущности, выполняющей программу во время своей активности
- Класс **Thread** реализует интерфейс **Runnable**.
- Интерфейс **Runnable** объявляет всего один метод – **public void run();**

```
public class MyRunnable implements Runnable {
    public void run() {
        // некоторое долгое вычисление
        long sum=0;
        for (int i=0; i<1000; i++) {
            sum+=i;
        }
        System.out.println(sum);
    }
}
```

```
Runnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
```

Работа с потоками

Для работы с потоками и для управления ими существуют следующие стандартные функции:

- `void start()` – запуск выполнения потока;
- `void stop()` – прекращение выполнения потока;
- `void suspend()` – приостанавливает выполнение потока;
- `void resume()` – возобновляет выполнение потока;
- `static void sleep (long millis)` – приостанавливает выполнение потока как минимум на millis миллисекунд;
- `static void yield()` – приостанавливает выполнение потока, предоставляя возможность выполнять другие потоки;
- `public final void join()` – ожидание безусловного завершения потока, для которого вызывается метод завершения потока или тайм-аута с более точным контролем времени
- `public final synchronized void join(long millis)` – ожидание завершения потока или истечения заданного числа миллисекунд (в зависимости то того, что произойдёт

Группы потоков

- Каждый поток принадлежит к некоторой группе
- Потоки делятся на группы потоков в целях безопасности (ограничение возможностей, чтобы потоки не мешали друг другу)
- Группа потоков может входить в состав другой группы (иерархия)
- Потоки внутри группы могут изменять другие потоки, входящие в ту же группу
- Поток не может модифицировать потоки за пределами своей собственной и всех подчинённых групп
- Ограничения, накладываемые на потоки, входящие в группу, описываются объектом **ThreadGroup**
- Группа может задаваться в конструкторе потока
- По умолчанию, каждый новый поток помещается в ту же группу, в которую входит его поток-создатель.
- После завершения потока, соответствующий объект удаляется из

Группы потоков

Конструкторы и методы объекта ThreadGroup:

- public **Thread (ThreadGroup group, String name)** – новый поток с заданным именем name (может быть равно null), принадлежащий конкретной группе
- public **ThreadGroup (String name)** – создаёт новую группу ThreadGroup
- public **ThreadGroup (ThreadGroup parent, String name)** – создаёт новую группу ThreadGroup с заданным именем, принадлежащую указанной parent группе
- public final String **ThreadGroup.getName()** – возвращает имя группы ThreadGroup
- public final **ThreadGroup ThreadGroup.getParent()** – возвращает родительскую группу ThreadGroup или null, если её не существует.
- public final void **setDaemon (boolean daemon)** – устанавливает «демонический» статус группы/потока
- public **Thread.getThreadGroup –** метод, позволяющий узнать, к какой группе принадлежит некоторый поток.
- public final void **Thread.checkAccess() –** метод, позволяющий проверить допустима ли модификация потока
- public final synchronized void **setMaxPriority(int maxPri) –** метод, задающий максимальный приоритет группы

Приоритеты потоков

- Процедура квантования времени поддерживает приоритеты (priority) задач
- Приоритет для задачи представляется целым числом
- Чем больше число – тем выше приоритет
- Поток с более высоким приоритетом получает большее количество квантов времени на исполнение

Для работы с приоритетами существуют такие основные методы класса Thread:

getPriority(), setPriority()

а так же объявлены три константы:

MIN_PRIORITY, MAX_PRIORITY, NORM_PRIORITY

Демон-потоки. Демон-группы

```
public class ThreadTest implements Runnable {
  // Отдельная группа, в которой будут
  // находится все потоки ThreadTest
  public final static ThreadGroup GROUP =
      new ThreadGroup("Daemon demo");
  // Стартовое значение
  private int start;
  public ThreadTest(int s) {
      start = (s % 2 == 0) ? s : s+1;
      new Thread(GROUP, this, "Thread "+start).start();
  }
  public void run() {
    try {
        // Начинаем обратный отсчет
        for (int i=start; i>0; i--) {
            Thread.sleep(300);
            // По достижению середины порождаем
            // поток с половинным новый значением
            // начальным
            if (start > 2 && i == start/2)
                new ThreadTest(i);
        }
    }
    catch (InterruptedException e) {}
  }
  public static void main(String s[]) {
      new ThreadTest(16);
      new DaemonDemo();
  }
}
```

```
class DaemonDemo extends Thread {
  public DaemonDemo() {
      super("Daemon demo thread");
      setDaemon(true);
      start();
  }
  public void run() {
    Thread threads[] = new Thread[10];
    try {
        while (true) {
            // Получаем набор всех потоков из
            // тестовой группы
            int count = ThreadTest.GROUP.activeCount();
            if (threads.length < count)
                threads = new Thread[count+10];
            count = ThreadTest.GROUP.enumerate(threads);
            // Распечатываем имя каждого потока
            for (int i=0; i<count; i++)
                System.out.print(threads[i].getName() + ", ");
            System.out.println();
            Thread.sleep(300);
        }
    }
    catch (InterruptedException e) {}
  }
}
```

Демон-потоки. Демон-группы

Результатом программы будет:

```
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16, Thread 8,
Thread 16, Thread 8,
Thread 16, Thread 8,
Thread 16, Thread 8,
Thread 16, Thread 8,
Thread 16, Thread 8, Thread 4,
Thread 16, Thread 8, Thread 4,
Thread 8, Thread 4
```

Разделяемые ресурсы

- Виртуальна машин поддерживае основное данны
  я (**main storage**), а в котором хранилище значения всех переменных
  сохраняются и
- Для каждого потока создается собственна рабоча памят
  которой используется всеми потоками (**working memory**), в которую передается использование копируются
  значения всех переменных.

volatile поля всегда читаются/записываются из/в основной памяти.

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>
  }
```

Блокировки

- Только один поток в один момент времени может вызвать синхронизированный метод и
- установить блокировку на некоторый объект
- Попытка блокировки уже заблокированного объекта приводит к останову потока до момента разблокирования данного объекта
- Планировщик потоков периодически активизирует потоки, ожидающие снятия блокировки
- Наличие блокировки не запрещает всех остальных действий с объектом
- Блокировка объект снимается автоматически при прекращении выполнения синхронизированного метода (исключительная ситуация)
- На объект возможно наложить блокировку и вызвать несколько синхронизированных методов
- Каждый объект имеет счётчик блокировок (**lock count**)
- Поток может наложить блокировку на несколько объектов одновременно
- Блокируется только объект, но не метод.

Взаимные блокировки

Взаимная блокировка (англ. **deadlock**) — ситуация в многопоточном программировании, при которой несколько потоков находятся в состоянии бесконечного ожидания ресурсов, захваченных самими этими потоками.

Поток 1	Поток 2
Необходимо захватить объект A и B, начинает с A	Необходимо захватить объект A и B, начинает с B
Блокирует объект A	
	Блокирует объект B
	Ожидает освобождения объекта A
Ожидает освобождения объекта B	
Взаимная блокировка	

Пример простой взаимной блокировки

- В Java нет никаких средств распознавания или предотвращения ситуаций **deadlock**.
- Также нет способа перед вызовом синхронизированного метода узнать, ли заблокирован уже объект другим потоком.

Методы класса Object (notify, wait, notifyAll)

- Каждый объект в Java имеет набор 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 группы объекта
- Метод **notifyAll()** пробуждает все потоки из wait-set группы объекта
- Любой из этих методов может быть вызван потоком у объекта только после установления блокировки на этот объект
- Потоки, после вызова метода **wait()** снимают все блокировки.
- После вызова метода **notify()** или **notifyAll()** потоки пытаются восстановить ранее снятые блокировки

Запрещенные действия над потоками. Прерывание потока

- **Thread.suspend(), Thread.resume()** – использование данных методов приводит к увеличению количества взаимных блокировок
- **Thread.stop()** – использование приводит к возникновению повреждённых объектов

- 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)

Если поток выполняет методы **wait(), sleep, join(),** прерывание потока методом **interrupt ()** приводит:

- Поток прерывает своё выполнение с выбросом исключения **InterruptedException**

Литература

- Курс лекций МФТИ «Программирование на Java»
- Дж. Гослинг – Язык программирования Java. 3-е издание
- http://www.intuit.ru/department/pl/javapl/12/1.html

Thank you!