Теория и технология программирования

Программирование на языке Java

Лекция 12. Основы многопоточного программирования

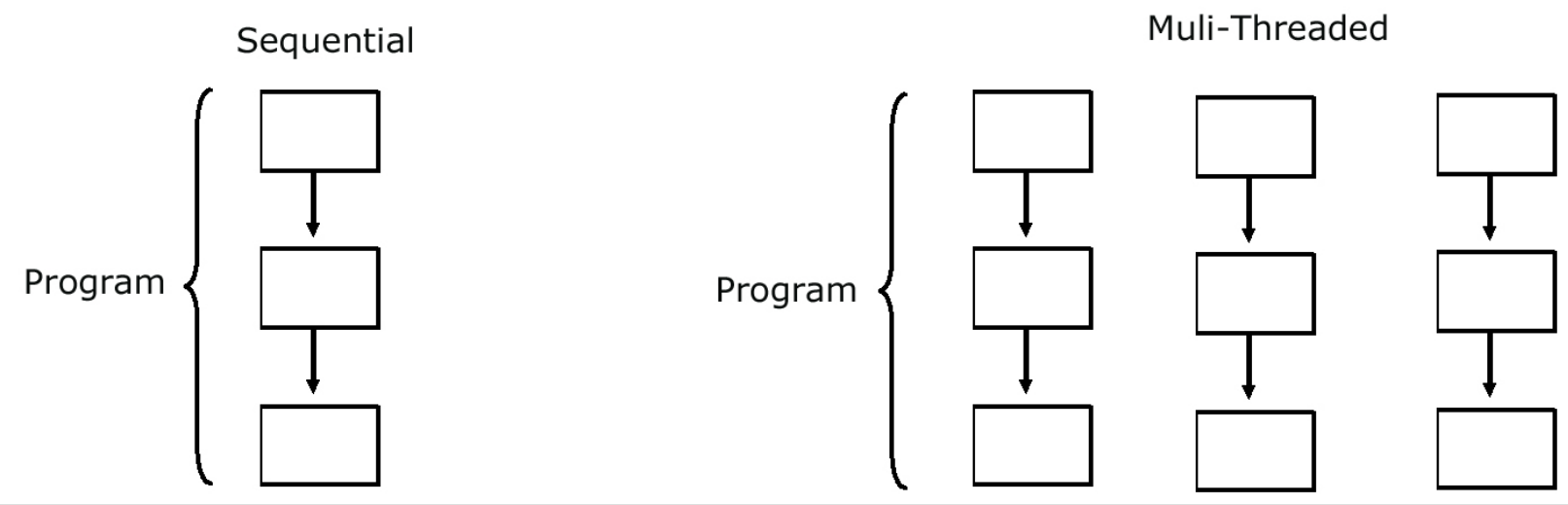
Глухих Михаил Игоревич, к.т.н., доц.
mailto: glukhikh@mail.ru

Определение процесса

- ***Процесс*** (process) – выполняющаяся программа, получающая от операционной системы собственное адресное пространство
  - ОС сама распределяет память между процессами (у каждого процесса она своя)
  - Программа также своя для каждого процесса
  - Если процессоров несколько, каждый из них может выполнять свой процесс
  - Если процессоров недостаточно, они выполняют то один процесс, то другой
  - Существуют способы взаимодействия между процессами

Определение потока

- ***Поток*** (thread) – часть многопоточной программы, выполняемая одновременно с другими такими же частями в одном адресном пространстве
  - у всех потоков память одна
  - программа также одна на все потоки



Аналогичные названия потока

- нить
- легковесный процесс
- подпроцесс
- тред

Диспетчеризация потоков

- Пусть имеется два процесса, А и Б
- Процесс А включает три потока – А1, А2, А3
- Процесс Б включает один поток – Б1
- Пример диспетчеризации для ПК с двумя процессорами (X и Y)
  - в интервал времени 0-100 мс процессор X выполняет поток А1, процессор Y выполняет поток Б
  - в интервал времени 100-200 мс процессор X выполняет поток А2, процессор Y выполняет поток А3
  - далее все повторяется
  - переключение между потоками не должно быть слишком частым

Приоритет потока

- Определяет частоту выполнения потока по сравнению с другими потоками
- Используется при выборе очередного потока на исполнение
- Пример – поток A1 имеет приоритет 10, поток A2 имеет приоритет 1
  - это может означать, что из 1 секунды 900 мс выполняется A1, 100 мс – A2
  - точный механизм зависит от ОС

Правила переключения между потоками

- Учет приоритета
  - вариант А – всегда выбирается один из потоков, имеющий наивысший приоритет (абсолютный приоритет)
  - вариант Б – потоку с большим приоритетом просто достается больше времени (относительный приоритет)
- Режим ожидания
  - любой поток может сказать, что он чего-либо ждет (например, завершения какого-то еще потока, или завершения определенного интервала времени)
  - пока поток в режиме ожидания, он не выполняется вне зависимости от его приоритета

Зачем нужны потоки

- Иногда важно, чтобы программа выполняла какие-то действия «условно одновременно»
  - Например, одновременно реагировала на действия пользователя в графическом интерфейсе и вела какие-то расчеты
  - Или принимала данные с нескольких других компьютеров, параллельно ведя расчеты
- Кроме этого, потоки дают возможность использовать наличие нескольких процессоров
  - Если в программе всего один поток, он будет выполняться на одном процессоре, остальные будут простаивать

Реализация потоков в Java

- Имеется встроенная поддержка потоков на уровне языка
  - интерфейс Runnable, класс Thread
  - ключевые слова volatile, synchronized
- При запуске любой программы создается так называемый **главный** поток (выполняющий функцию main)
  - главный поток может создавать другие потоки
  - они, в свою очередь, могут создавать дополнительные потоки
- Существуют дополнительные потоки сборщика мусора (Garbage Collector Threads)
- Используется схема с абсолютными приоритетами

Интерфейс Runnable

- То, что можно выполнить
- Содержит единственный метод:
  `void run();`
- Может использоваться для описания действий, которые должны быть выполнены в отдельном потоке

Класс Thread

- Выполняемый поток
- Реализует Runnable:
  class Thread implements Runnable
- Метод run() по умолчанию не делает ничего – должен быть переопределен в производном классе
- Для запуска потока на исполнение служит метод start()

Обратите внимание!

- Если мы в потоке А вызовем метод Thread.run(), действия будут выполнены в том же потоке А
- Если мы в потоке А вызовем метод Thread.start(), будет создан поток Б, который будет выполнять метод run(), а поток А продолжит выполнение со следующей после вызова Thread.start() команды

Два способа создания своих потоков

```
// Способ 1 – на основе Runnable
// Описание действий потока
class MyRunnable implements Runnable {
    public void run() { … }
}
// Запуск потока
// …
Runnable runnable = new MyRunnable();
// Создаем поток на основе выполняемого объекта
Thread thread = new Thread(runnable);
thread.start();
// …
```

Два способа создания своих потоков

```
// Способ 2 – на основе Thread
// Описание действий потока
class MyThread extends Thread {
   @Override
   public void run() { … }
}
// Запуск потока
// …
// Создаем собственный поток
Thread thread = new MyThread();
thread.start();
// …
```

Сравнение двух способов

- Способ на основе Thread требует написания меньшего количества кода
- Кроме того, в собственном потоке можно переопределить и другие методы класса Thread (впрочем, это редко требуется)
- Класс на основе Runnable, в отличие от класса на основе Thread, можно унаследовать от какого-либо еще класса (и это требуется достаточно редко)

Пример – часы

```
public class PeriodicThread extends Thread {
    private final int period;
    private final ActionListener listener;
    public PeriodicThread(int period, ActionListener listener)
{
        this.period = period;
        this.listener = listener;
    }
}
```

Пример – часы

```
public class PeriodicThread extends Thread {
    @Override
    public void run() {
        for (;;) {
            try {
                Thread.sleep(period);
            } catch (InterruptedException ex) {
                return;
            }
            if (listener!=null)
                listener.actionPerformed(
                    new ActionEvent(this, 0, "Periodic"));
} } }
```

Демонстрация примера

- См.

Обзор методов Thread

- Thread() – простой конструктор потока (метод run должен быть переопределен)
- Thread(Runnable r) – конструктор на основе Runnable (метод run переопределять не надо)
- void run() – метод, содержащий выполняемые потоком действия
- void start() – создать новый поток и запустить на исполнение метод run()
- static void sleep(long ms) – остановить выполнение **текущего** потока и ждать указанное число миллисекунд
- void join(), void join(long ms) – остановить выполнение **текущего** потока и ждать завершения указанного потока; если указано время – ждать не более этого времени
- void interrupt() – прервать поток (заканчивает все состояния ожидания путем InterruptedException)

Обзор методов Thread

- static Thread currentThread() – возвращает ссылку на текущий поток
- getName(), setName() – получение и установка имени потока
- isAlive() – жив ли поток
- getState() – вернуть состояние потока, из
  - NEW – создан, но не запущен
  - RUNNABLE – выполняется
  - BLOCKED – ожидает ресурсов или событий
  - WAITING, TIMING_WAITING – ожидает другой поток
  - TERMINATED - завершен

Пример на многопроцессорную работу

- Требуется определить количество простых чисел в интервале от 2 до N
- При большом N (более миллиона) решение задачи требует существенного времени даже на современном ПК

Базовое решение (однопоточное)

```
final List<Integer> primes = new ArrayList<Integer>();
primes.add(2);
for (int i=3; i<20000000; i+=2) {
    boolean isPrime = true;
    for (int prime: primes) {
        if (prime * prime > i) break;
        if (i % prime == 0) {
            isPrime = false; break;
        }
    }
    if (isPrime) primes.add(i);
}
System.out.println("Total primes found: " + primes.size());
```

Распараллеливание

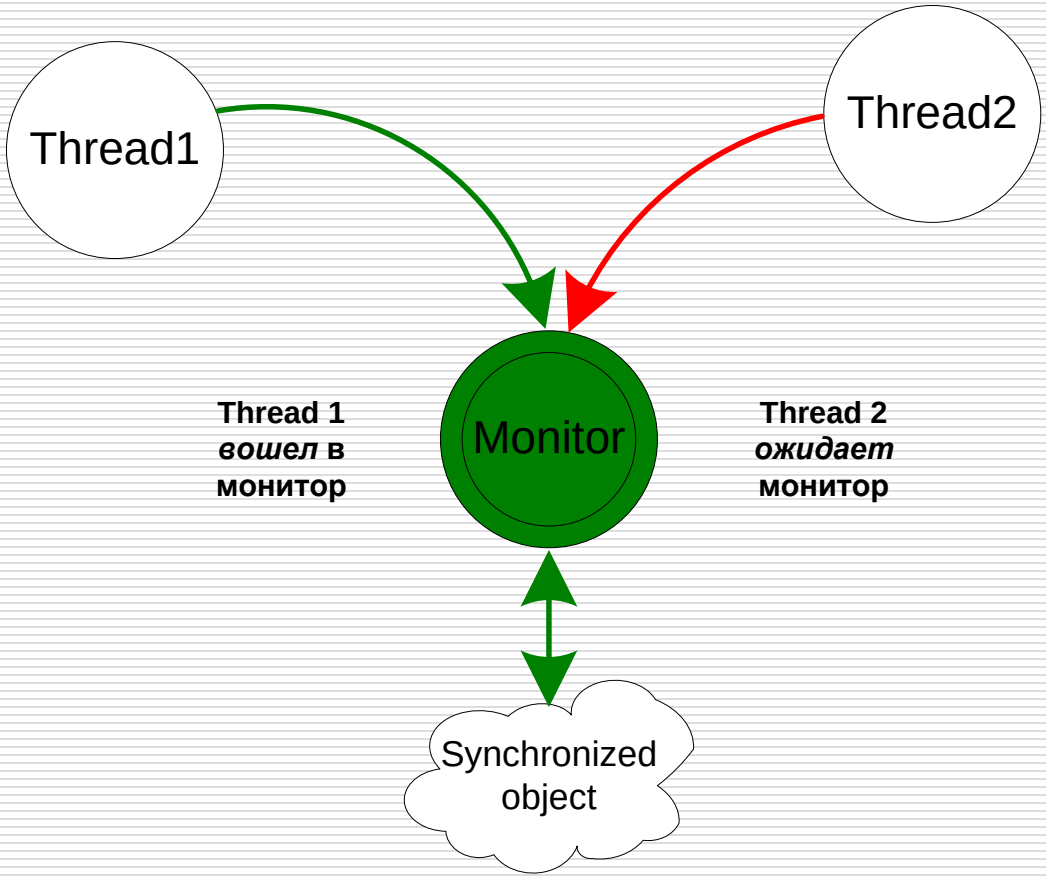
- Как разделить работу между двумя нитями примерно пополам?

Распараллеливание

- Как разделить работу между двумя потоками примерно пополам?
- Одно из решений – первый поток проверяет числа 3, 7, 11, 15, ...,
  второй поток
  проверяет числа 5, 9, 13, 17, ...
- Проблема – оба потока должны работать с общим списком простых чисел – что если один поток изменит этот список, пока второй поток его читает?

Синхронизация по ресурсам

- Если один поток работает с некоторым объектом, второй поток не может работать с ним в это же время



Синхронизация по ресурсам – язык Java

- synchronized методы – одновременно вызываются только одной нитью
  - public synchronyzed int get() { ... }
- synchronized(obj) блоки – на время выполнения блока объект obj блокируется

Синхронизация по ресурсам - проблемы

- Следует понимать, что каждый раз, когда поток пытается получить доступ к занятому ресурсу, происходит переключение на другой поток
- Если к ресурсу обращаются часто, переключения будут занимать очень много времени

Распараллеливание

- Обратите внимание, что при проверке числа N на простоту нас интересуют только простые числа в пределах до квадратного корня из N
- Давайте еще до распараллеливания найдем все простые числа в этих пределах – времени потребуется сравнительно мало
- А потом мы создадим два отдельных списка, и у нитей не будет общего ресурса

Поток поиска целых чисел

```
public class PrimeChecker extends Thread {
    private int step, last, current;
    private final List<Integer> primes;
    private boolean checkCurrent() {
        for (int prime: primes) {
            if (prime*prime > current)
                break;
            if (current % prime == 0)
                return false;
        }
        return true;
    }
}
```

Поток поиска целых чисел

```
public class PrimeChecker extends Thread {
    private int step, last, current;
    private final List<Integer> primes;
    public PrimeChecker(int start, int step,
            int last, List<Integer> primes) {
        this.step = step;
        this.last = last;
        this.primes = primes;
        current = start;
    }
    public List<Integer> getPrimes() {
        return primes;
    }
}
```

Поток поиска целых чисел

```
public class PrimeChecker extends Thread {
    private int step, last, current;
    private final List<Integer> primes;
    @Override
    public void run() {
        while (current <= last) {
            final boolean isPrime = checkCurrent();
            if (isPrime) primes.add(current);
            current += step;
        }
        System.out.println("Thread " + getName() +
                " has finished work");
    }
}
```

Главная функция

```
final List<Integer> list = new ArrayList<Integer>();
list.add(2);
int threadNumber = 4;
int limit = 20000000;
int last = (int)Math.sqrt(limit)+1;
final PrimeChecker firstChecker = new
    PrimeChecker(3,2,last,list);
firstChecker.setName("Base checker");
firstChecker.run();
if (last % 2 == 0) last++;
final PrimeChecker[] checkers=new PrimeChecker[threadNumber];
final List[] copies = new List[threadNumber];
```

Главная функция

```
for (int i=0; i<threadNumber; i++) {
    final List<Integer> listCopy =
       new ArrayList<Integer>(list);
    checkers[i] = new
       PrimeChecker(last+2*i,2*threadNumber,limit,listCopy);
    checkers[i].setName("Checker #" + i);
    copies[i] = listCopy;
}
for (int i=1; i<threadNumber; i++) checkers[i].start();
checkers[0].run();
```

Главная функция

```
try {
    for (int i=1; i<threadNumber; i++) checkers[i].join();
} catch (InterruptedException ex) {}
int total = list.size();
for (int i=0; i<threadNumber; i++)
    total += (copies[i].size() - list.size());
System.out.println("Total primes found: " + total);
```

Статистика (на ПК с тремя ядрами)

- 1 поток – 30 секунд
- 2 потока – 15 секунд
- 3 потока – 15 секунд
- >3 потоков – 10-11 секунд

Synchronized / Volatile

- Synchronized = mutual exclusion + write in thread A / read from thread B
- Volatile = only write in thread A / read from thread B
- Example: StopThread

Executors / Tasks

- Executor / ExecutorService = умеет управлять исполнением чего-либо, обычно содержит внутри какие-то нити
  - executor.execute(runnable)
  - Executors.newSingleThreadExecutor()
  - Executors.newCachedThreadPool()
  - Executors.newFixedThreadPool()
- task = Runnable / Callable
  - Future<T> submit(Callable)
  - Future<T>: get(), cancel(), isDone()…

Concurrent collections

- Vector / Hashtable (deprecated)
- Collections.synchronized…
- ConcurrentHashMap
- BlockingQueue
- CopyOnWriteArrayList / Set