Sistem Terdistribusi

TIK-604

Husni.trunojoyo.ac.id

Pemrograman Konkuren dengan Thread Java

Topik Praktik (Belajar Mandiri)

Husni

husni@trunojoyo.ac.id

Garis Besar Bahasan

- Motivasi
  - Perbandingan pemrograman konkuren & paralel
- Pendekatan dasar
  - Membuat suatu task list dengan Executors.newFixedThreadPool
  - Menambahkan task ke list dengan taskList.execute(someRunnable)
- Variasi pada tema
  - Kelas tersendiri yang mengimplementasikan ***Runnable***
  - Kelas utama mengimplementasikan ***Runnable***
  - *Inner classes* yang mengimplementasikan ***Runnable***
  - Ekspresi Lambda
- Topik terkait
  - Race condition dan sinkronisasi
  - Metode-metode penting terkait thread
  - Topik lanjutan dalam konkurensi

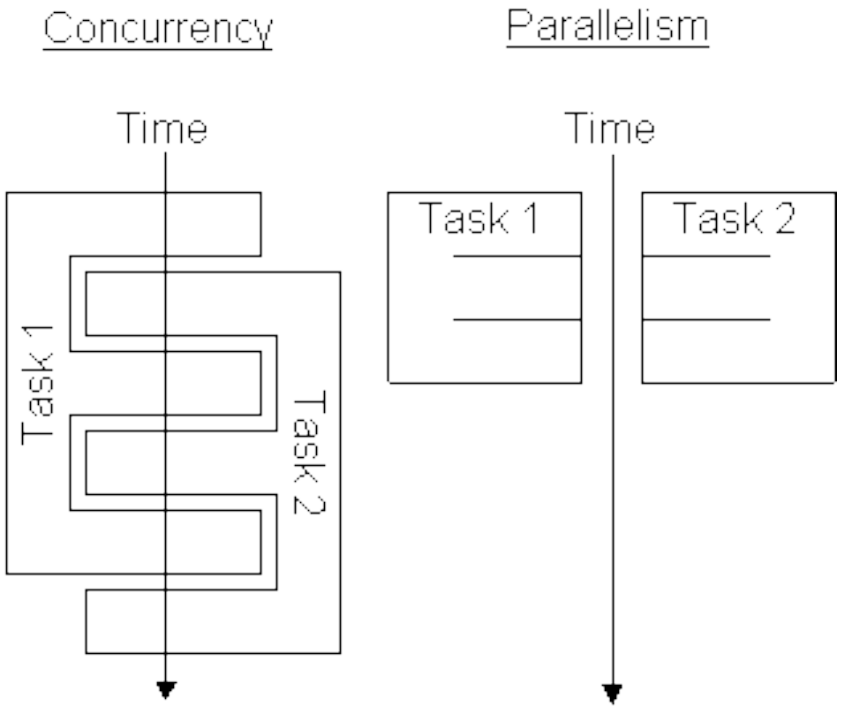
Ikhtisar

Motivasi Pemrograman Konkuren

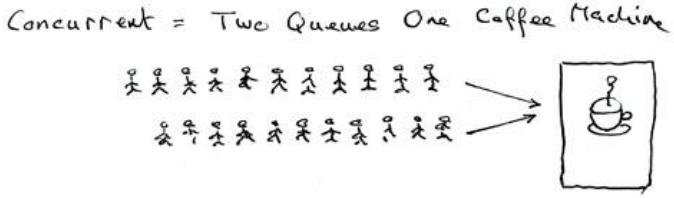
- Pro
  - Untung, bahkan pada sistem berprocessor tunggal
    - Efisiensi
      - Download file-file (data) di jaringan lebih cepat
    - Kenyamanan
      - Suatu ikon jam (di desktop / taskbar misalnya)
    - Aplikasi banyak client
      - Server HTTP, Server SMTP
  - Sebagian besar komputer mempunyai banyak processor (atau core)
    - Temukan dengan Runtime.getRuntime().availableProcessors()
- Kontra
  - Jelas lebih sulit men-*debug* dan merawatnya daripada aplikasi berthread tunggal

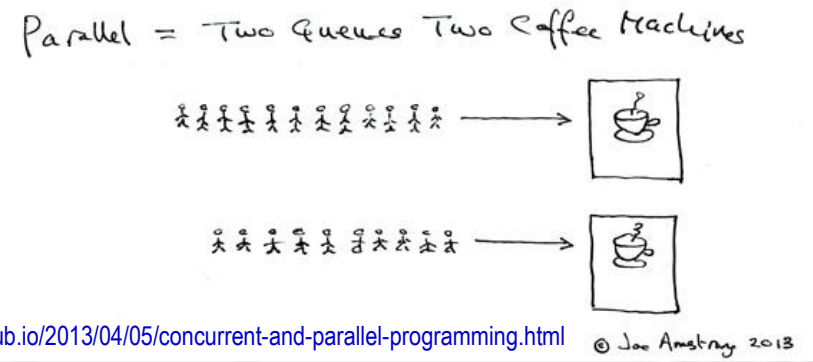
Pemrograman Konkuren vs. Paralel

- Konkuren
  - Task-task yang *overlap* dalam waktu
    - Sistem mungkin menjalankan beberapa tugas secara paralel pada banyak processor atau pergantian giliran antar tugas pada processor yang sama.
- Paralel
  - Beberapa tugas berjalan pada waktu sama di processor berbeda



http://www.javaworld.com/article/2076774/java-concurrency/programming-java-threads-in-the-real-world--part-1.html





http://joearms.github.io/2013/04/05/concurrent-and-parallel-programming.html

Thread Java (Konkuren) vs. Framework Fork/Join (Paralel)

- Menggunakan thread
  - Saat task relatif besar dan *self-contained*
  - Biasanya ketika proses menunggu sesuatu, sehingga menguntungkan bahkan jika ada hanya satu processor
  - Dibahas di sesi ini
    - Diperlukan bahkan dalam Java 8 yang telah mempunyai stream paralel
- Menggunakan fork/join atau stream paralel
  - Ketika task besar di permulaan tetapi dapat dipecah secara berulang menjadi potongan-potongan lebih kecil, kemudian digabungkan untuk hasil akhirnya.
  - Tidak mendapatkan keuntungan jika hanya ada satu processor
  - Dibahas di lain waktu.
    - Versi paralel dari Stream menggunakan framework fork/join dapat menangani sebagian besar situasi saat kita ingin menggunakan fork/join, dan secara dramatis lebih simpel daripada menggunakan fork/join secara eksplisit. Sebagian besar pengembang Java 8 dapat melewati topik fork/join dan langsung fokus pada stream paralel.

Langkah-langkah Dasar Pemrograman Konkuren

Langkah-langkah Pemrograman Konkuren

- Membuat suatu task list

```
ExecutorService taskList =
  Executors.newFixedThreadPool(poolSize);
```

  - **poolSize** adalah jumlah maksimum dari thread simultan. Pada banyak aplikasi, nilainya lebih tinggi daripada jumlah task, sehingga setiap task mempunyai thread tersendiri.
  - Ada jenis lain dari thread pool, tetapi thread pool “tetap” ini lebih mudah dan umum.

- Tambahkankan task Runnable ke list tersebut

```
taskList.execute(someRunnable);
```

Variasi Penambahan Runnable ke Task List

- Membuat kelas tersendiri yang mengimplementasikan Runnable
  - `taskList.execute(new MySeparateRunnableClass(…));`
- Meminta kelas yang ada untuk mengimplementasikan Runnable
  - `taskList.execute(this);`
- Menggunakan *inner class*
  - `taskList.execute(new MyInnerRunnableClass(…));`
    - Ini dapat berupa *inner class* bernama atau anonim
- Menggunakan lambda
  - `taskList.execute(() -> codeForRunMethod());`

Pendekatan Satu: Kelas Tersendiri yang Mengimplementasikan Runnable

Mekanisme Thread Satu: Kelas Runnable Tersendiri

- Membuat kelas yang mengimplementasikan Runnable
  - Tidak diperlukan pernyataan import : Runnable ada dalam java.lang
- Letakkan aksi untuk dikerjakan dalam metode run

```
public class MyRunnable implements Runnable {
  public void run() { someBackgroundTask(); }
}
```

- Membuat satu *instance* dari kelas tersebut
  - Atau banyak *instance* jika kita ingin terbentuk banyak *thread*
- Kirimkan *instance* ke ExecutorService.execute

```
taskList.execute(new MyRunnable(…));
```

  - Jumlah thread simultan tidak akan melebihi ukuran maksimum dari pool.

Kelas Runnable Tersendiri: Kode Template

```
public class MainClass extends SomeClass {
  ...
  public void startThreads() {
    int poolSize = ...;
    ExecutorService taskList = Executors.newFixedThreadPool(poolSize);
    for(int i=0; i<something; i++) {
      taskList.execute(new SomeTask(...));
    }
  }
}
```

```
public class SomeTask implements Runnable {
  public void run() {
    // kode yang berjalan di latar belakang
  }
}
```

Mekanisme Thread Satu: Contoh

```
public class Counter implements Runnable {
  private final App1 mainApp;
  private final int loopLimit;

  public Counter(App1 mainApp, int loopLimit) {
    this.mainApp = mainApp;
    this.loopLimit = loopLimit;
  }

  public void run() {
    for(int i=0; i<loopLimit; i++) {
      String threadName = Thread.currentThread().getName();
      System.out.printf("%s: %s%n", threadName, i);
      mainApp.pause(Math.random());
    }
  }
}
```

Mekanisme Thread Satu: Contoh (Lanj.)

```
public class App1 extends SomeClass {
  public App1() {
    ExecutorService taskList = Executors.newFixedThreadPool(100);
    taskList.execute(new Counter(this, 6));
    taskList.execute(new Counter(this, 5));
    taskList.execute(new Counter(this, 4));
    taskList.shutdown();
  }

  public void pause(double seconds) {
    try {
      Thread.sleep(Math.round(1000.0 * seconds));
    } catch (InterruptedException ie) { }
  }
}
```

Metode shutdown berarti bahwa task list tersebut tidak akan lagi menerima task baru (via execute). Tasks yang sudah dalam antrian akan masih berjalan. Biasanya tidak perlu memanggil shutdown, tapi pada kasus ini, kita ingin program untuk exit setelah tasks selesai. Jika kita tidak memanggil shutdown di sini, kita harus mematikan proses dengan Control-C (command line) atau klik tombol merah (Eclipse), karena thread latar belakang masih terus berjalan, menunggu task baru ditambahkan ke antrian (*queue*).

Mekanisme Thread Satu: Contoh (Lanj.)

```
public class App1Test {
  public static void main(String[] args) {
    new App1();
  }
}
```

Mekanisme Thread Satu: Hasil

```
pool-1-thread-1: 0
pool-1-thread-2: 0
pool-1-thread-3: 0
pool-1-thread-2: 1
pool-1-thread-2: 2
pool-1-thread-1: 1
pool-1-thread-3: 1
pool-1-thread-2: 3
pool-1-thread-3: 2
pool-1-thread-1: 2
pool-1-thread-1: 3
pool-1-thread-1: 4
pool-1-thread-3: 3
pool-1-thread-2: 4
pool-1-thread-1: 5
```

Pro & Kontra Pendekatan Kelas Tersendiri

- Keuntungan
  - Loose coupling
    - Dapat mengubah potongan-potongan secara bebas
    - Dapat menggunakan-ulang kelas Runnable dalam lebih dari satu aplikasi
  - Pengiriman argumen
    - Jika kita ingin thread-thread berbeda melakukan hal berbeda, kita lewatkan args ke konstruktor, yang menyimpannya dalam variabel instance yang digunakan oleh metode run.
  - Bahaya race conditions
    - Kita biasanya menggunakan pendekatan ini saat tidak ada data yang dishare antar thread, sehingga tidak perlu disinkronisasi.
- Kerugian
  - Sulit mengakses aplikasi utama
    - Jika kita ingin memanggil metode dalam kelas utama, harus
      - Mengirimkan referensi ke aplikasi utama ke konstruktor, yang menyimpannya
      - Membuat metode dalam aplikasi utama menjadi publik cakupannya

Pendekatan Dua: Aplikasi Utama Mengimplementasikan *Runnable*

Mekanisme Thread Dua: Aplikasi Utama Mengimplementasikan Runnable

- Memiliki kelas utama yang mengimplementasikan Runnable
  - Letakkan aksi dalam metode run dari kelas yang ada

```
public class MyClass extends Something implements Runnable {
  ...
  public void run() {
    ...
  }
}
```

- Kirimkan instance dari kelas utama untuk dieksekusi

```
taskList.execute(this);
```

Kelas Utama Mengimplementasikan Runnable: Perbedaan Utama vs. Kelas Tersendiri

- Kabar baik
  - run dapat dengan mudah memanggil metode dalam kelas utama karena ada di dalam kelas tersebut

- Kabar buruk
  - Jika run mengakses suatu *shared data* (*instance variables*), kita harus perhatikan konfliknya (*race condition*)
  - Tanpa konstruktor, sehingga sangat sulit mengkustom bagaimana setiap thread berjalan, sehingga setiap task dimulai dengan cara yang sama

Aplikasi Utama Mengimplementasikan Runnable: Kode Template

```
public class ThreadedClass extends AnyClass implements Runnable {
  public void run() {
    // kode yang akan berjalan di latar belakang
  }

  public void startThreads() {
    int poolSize = ...;
    ExecutorService taskList = Executors.newFixedThreadPool(poolSize);
    for(int i=0; i<someSize; i++) {
      taskList.execute(this);
    }
  }
  ...
}
```

Mekanisme Thread Dua: Contoh

```
public class App2 extends SomeClass implements Runnable {
  private final int loopLimit;

  public App2(int loopLimit) {
    this.loopLimit = loopLimit;
    ExecutorService taskList = Executors.newFixedThreadPool(100);
    taskList.execute(this);
    taskList.execute(this);
    taskList.execute(this);
    taskList.shutdown();
  }

  private void pause(double seconds) {
    try {
      Thread.sleep(Math.round(1000.0 * seconds));
    } catch (InterruptedException ie) { }
  }
```

Kelas berlanjut ke slide berikutnya

Mekanisme Thread Dua: Contoh (Lanj.)

```
  public void run() {
    for(int i=0; i<loopLimit; i++) {
      String threadName = Thread.currentThread().getName();
      System.out.printf("%s: %s%n", threadName, i);
      pause(Math.random());
    }
  }
}
```

Mekanisme Thread Dua: Contoh (Lanj.)

```
public class App2Test {
  public static void main(String[] args) {
    new App2(5);
  }
}
```

Mekanisme Thread Dua: Hasil

```
pool-1-thread-3: 0
pool-1-thread-1: 0
pool-1-thread-2: 0
pool-1-thread-2: 1
pool-1-thread-3: 1
pool-1-thread-3: 2
pool-1-thread-1: 1
pool-1-thread-2: 2
pool-1-thread-3: 3
pool-1-thread-2: 3
pool-1-thread-1: 2
pool-1-thread-3: 4
pool-1-thread-1: 3
pool-1-thread-2: 4
pool-1-thread-1: 4
```

Pro & Kontra Pendekatan

- Keuntungan
  - Mudah mengakses aplikasi utama
    - Metode run ada di dalam aplikasi utama. Dapat mengakses metode publik atau privat atau variabel instance.
- Kekurangan
  - Ikatan ketat (*tight coupling*)
    - Metode run terikat erat ke aplikasi ini
  - Tidak dapat melewatkan argumen ke metode run
    - Jadi, kita dapat memulai thread tunggal saja (umum), atau semua thread melakukan tugas yang sangat mirip
  - Bahaya *race condition*
    - Kita biasanya menggunakan pendekatan ini secara khusus karena ingin mengakses data di aplikasi utama. Jadi, jika metode run memodifikasi data bersama, kita harus melakukan sinkronisasi.

Pendekatan Tiga: *Inner Class* Mengimplementasikan *Runnable*

Mekanisme Thread Tiga: *Runnable Inner Class*

- Memiliki inner class yang mengimplementasikan Runnable
  - Letakkan aksi dalam metode run dari *inner class*

```
public class MyClass extends Whatever {
  …
  private class InnerClass implements Runnable {
    public void run() {
      ...
    }
  }
}
```

- Kirimkan *instances* dari *inner class* untuk dieksekusi

```
taskList.execute(new InnerClass(...));
```

Runnable Inner Class: Kode Template

```
public class MainClass extends AnyClass {
  public void startThreads() {
    int poolSize = ...;
    ExecutorService taskList = Executors.newFixedThreadPool(poolSize);
    for(int i=0; i<someSize; i++) {
      taskList.execute(new RunnableClass(...));
    }
  }
  ...
  private class RunnableClass implements Runnable {
    public void run() {
      // Kode yang akan dijalankan di latar belakang
    }
  }
}
```

Variasi Minor: Inner Class Anonim

```
public class MainClass extends AnyClass {
  public void startThreads() {
    int poolSize = ...;
    ExecutorService taskList =
      Executors.newFixedThreadPool(poolSize);
    for(int i=0; i<someSize; i++) {
      taskList.execute(new Runnable() {
        public void run() {
          ...
        }
      });
    }
  }
}
```

Mekanisme Thread Tiga: Contoh

```
public class App3 extends SomeClass {
  public App3() {
    ExecutorService taskList = Executors.newFixedThreadPool(100);
    taskList.execute(new Counter(6));
    taskList.execute(new Counter(5));
    taskList.execute(new Counter(4));
    taskList.shutdown();
  }

  private void pause(double seconds) {
    try {
      Thread.sleep(Math.round(1000.0 * seconds));
    } catch (InterruptedException ie) { }
  }
```

Kelas berlanjut pada slide berikutnya

Mekanisme Thread Tiga: Contoh (Lanj.)

```
  private class Counter implements Runnable {  // Inner class
    private final int loopLimit;

    public Counter(int loopLimit) {
      this.loopLimit = loopLimit;
    }

    public void run() {
      for(int i=0; i<loopLimit; i++) {
        String threadName = Thread.currentThread().getName();
        System.out.printf("%s: %s%n", threadName, i);
        pause(Math.random());
      }
    }
  }
}
```

Kita dapat pula menggunakan inner class anonim. Ini tidak cukup beda untuk menjamin contoh sendiri di sini, khususnya karena kita memper-lihatkan contoh pada bagian penanganan kejadian.

Mekanisme Thread Tiga: Contoh (Lanj.)

```
public class App3Test {
  public static void main(String[] args) {
    new App3();
  }
}
```

Mekanisme Thread Tiga: Hasil

```
pool-1-thread-2: 0
pool-1-thread-1: 0
pool-1-thread-3: 0
pool-1-thread-3: 1
pool-1-thread-1: 1
pool-1-thread-1: 2
pool-1-thread-2: 1
pool-1-thread-3: 2
pool-1-thread-3: 3
pool-1-thread-1: 3
pool-1-thread-1: 4
pool-1-thread-1: 5
pool-1-thread-2: 2
pool-1-thread-2: 3
pool-1-thread-2: 4
```

Pro & Kontra Pendekatan

- Kelebihan
  - Mudah mengakses aplikasi utama.
    - Metode dalam inner class dapat mengakses suatu metode privat atau publik atau variabel *instance* dari *outer class*
  - Dapat melewatkan argumen ke metode run
    - Sebagaimana dengan kelas terpisah, kita mengirimkan argumen ke konstruktor, yang menyimpannya dalam variabel instance yang digunakan metode run.
- Kekurangan
  - *Tight coupling*
    - Metode run terikat ketat ke aplikasi ini
  - Bahaya *race condition*
    - Kita biasanya menggunakan pendekatan ini secara khusus karena ingin mengakses data di aplikasi utama. Jadi, jika run memodifikasi beberapa data bersama, harus dilakukan sinkronisasi.

Preview Pendekatan Empat: Ekspresi Lambda

Pratinjau Lambdas

- inner class Anonim

```
taskList.execute(new Runnable() {
  @Override
  public void run() {
    doSomeTask(…);
  }
});
```

- Lambda yang Ekuivalen

```
taskList.execute(() -> doSomeTask(…));
```

Lambda expressions akan dibahas disesi lain. Tutorial singkatnya dapat dilihat di : http://www.coreservlets.com/java-8-tutorial/

Rangkuman Pendekatan-Pendekatan

Pro & Kontra

- Kelas tersendiri yang mengimplementasikan Runnable
  - Tidak dapat dengan mudah mengakses data dalam kelas main (dan hanya data publik)
  - Dapat melewatkan argumen ke run (secara tidak langsung via konstruktor dan variabel instance)
  - Biasanya tidak khawatir dengan race conditions
- Kelas utama (main) yang mengimplementasikan Runnable
  - Dapat dengan mudah mengakses data dalam main class
  - Tidak dapat melewatkan argumen ke metode run
  - Harus khawatir terjadinya race conditions
- Inner class mengimplementasikan Runnable
  - Dapat dengan mudah mengakses data dalam main class
  - Dapat melewatkan argumen ke run (secara tk langsung via konstruktor dan variabel instance)
  - Harus khawatir terjadinya race conditions
- Lambdas
  - Dapat dengan mudah mengakses data dalam main class
  - Tidak dapat melewatkan argumen ke run (bukan variabel instance)
  - Harus khawatir terjadinya race condition.

Contoh: Template Server Jaringan Multithreaded

```
import java.net.*;
import java.util.concurrent.*;
import java.io.*;

public class MultithreadedServer {
  private int port;

  public MultithreadedServer(int port) {
    this.port = port;
  }

  public int getPort() {
    return(port);
  }
```

MultithreadedServer.java (Lanj.)

```
public void listen() {
   int poolSize = 100 * Runtime.getRuntime().availableProcessors();
   ExecutorService taskList = Executors.newFixedThreadPool(poolSize);
   try {
     ServerSocket listener = new ServerSocket(port);
     Socket socket;
     while(true) {  // Jalan sampai mati
       socket = listener.accept();
       taskList.execute(new ConnectionHandler(socket));
     }
   } catch (IOException ioe) {
     System.err.println("IOException: " + ioe);
     ioe.printStackTrace();
   }
  }
}
```

Slide berikutnya tentang pemrograman jaringan akan menjelaskan pemanfaatan ServerSocket & Socket. Tetapi ide dasarnya adalah server menerima koneksi dan kemudian meletakkannya dalam antrian tugas sehingga dapat ditangani thread di latar belakang.

ConnectionHandler.java

```
public class ConnectionHandler implements Runnable {
  private Socket socket;

  public ConnectionHandler(Socket socket) { this.socket = socket; }

  public void run() {
    try {  handleConnection(socket); }
    catch(IOException ioe) {
      System.err.println("IOException: " + ioe);
      ioe.printStackTrace();
    }
  }

  public void handleConnection(Socket socket) throws IOException{
    // Lakukan sesuatu dengan socket
  }
}
```

Race Conditions & Sinkronisasi

Race Conditions: Contoh

```
public class RaceConditionsApp implements Runnable {
  private final static int LOOP_LIMIT = 5;
  private final static int POOL_SIZE = 10;
  private int latestThreadNum = 0;

  public RaceConditionsApp() {
    ExecutorService taskList;
    taskList = Executors.newFixedThreadPool(POOL_SIZE);
    for (int i=0; i<POOL_SIZE; i++) {
      taskList.execute(this);
    }
  }
```

Race Conditions: Contoh (Lanj.)

```
public void run() {
  int currentThreadNum = latestThreadNum;
  System.out.println("Set currentThreadNum to " + currentThreadNum);
  latestThreadNum = latestThreadNum + 1;
  for (int i=0; i<LOOP_LIMIT; i++) {
    doSomethingWith(currentThreadNum);
  }
}

private void doSomethingWith(int threadNumber) {
  // Blah blah
}
```

- Apa yang salah dengan kode di atas?

Race Conditions: Hasil

- Output diharapkan

```
Set currentThreadNum to 0
Set currentThreadNum to 1
Set currentThreadNum to 2
Set currentThreadNum to 3
Set currentThreadNum to 4
Set currentThreadNum to 5
Set currentThreadNum to 6
Set currentThreadNum to 7
Set currentThreadNum to 8
Set currentThreadNum to 9
```

- Output diperoleh

```
Set currentThreadNum to 0
Set currentThreadNum to 1
Set currentThreadNum to 2
Set currentThreadNum to 3
Set currentThreadNum to 4
Set currentThreadNum to 5
Set currentThreadNum to 5
Set currentThreadNum to 7
Set currentThreadNum to 8
Set currentThreadNum to 9
```

Race Conditions: Solusi?

- Lakukan sesuatu dalam langkah tunggal

```
public void run() {
  int currentThreadNum = latestThreadNum++;
  System.out.println("Set currentThreadNum to " +
                     currentThreadNum);
  for (int i=0; i<LOOP_LIMIT; i++) {
    doSomethingWith(currentThreadNum);
  }
}
```

"Solusi" ini tidak membetulkan masalah.
Pada beberapa cara, malah makin parah.

Menengahi Pertarungan Sumber Daya Bersama

- Sinkronisasi bagian dari kode

```
synchronized(someObject) {
   code
}
```

- Interpretasi normal
  - Begitu suatu thread memasuki bagian kode tertentu, tidak ada thread lain boleh masuk sampai thread itu keluar
- Interpretasi lebih kuat
  - Setelah suatu thread memasuki bagian kode tertentu, tidak ada thread lain dapat memasuki suatu bagian kode yang disinkronkan enggunakan obyek “lock” yang sama.
    - Jika dua potong kode berkata “synchronized(blah)”, pertanyaannya apakah blah merupakan instance obyek yang sama.

Menengahi Pertarungan Sumber Daya Bersama

- Sinkronisasi seluruh metode

```
public synchronized void someMethod() {
  body
}
```

- Ini sama dengan

```
public void someMethod() {
  synchronized(this) {
    body
  }
}
```

Perbaikan Race Condition Sebelumnya

```
public void run() {
  int currentThreadNum;
  synchronized(this) {
    currentThreadNum = latestThreadNum;
    System.out.println("Set currentThreadNum to + currentThreadNum);
    latestThreadNum = latestThreadNum + 1;
  }
  for (int i=0; i<LOOP_LIMIT; i++) {
    doSomethingWith(currentThreadNum);
  }
}
```

Metode Penting dari Thread

Metode dalam Kelas Thread

- Thread.currentThread()
  - Memberikan instance dari Thread yang menjalankan kode saat ini
- Thread.sleep(milliseconds)
  - Meletakkan kode yang memangil ke sleep. Berguna untuk penantian *non-busy* dalam semua jenis kode, tidak hanya kode multithreaded. Harus menangkap InterruptedException, tetapi boleh langsung melepasnya:

```
try { Thread.sleep(someMilliseconds); }
catch (InterruptedException ie) { }
```

  - Lihat juga TimeUnit.SECONDS.sleep, TimeUnit.MINUTES.sleep, dll.
    - Idenya sama tetapi mengambil sleep time dalam satuan berbeda.
- someThread.getName(), someThread.getId()
  - Berguna untuk *printing/debugging*.

Metode dalam Kelas ExecutorService

- execute(Runnable)
  - Menambahkan Runnable ke antrian task
- shutdown
  - Mencegah lebih banyak task ditambahkan dengan execute (atau submit), tetapi mempersilakan task aktif diselesaikan.
- shutdownNow
  - Usaha untuk menghentikan task aktif. Tetapi pembuat task harus mampu membuatnya merespon dengan interupsi (yaitu catch InterruptedException), atau ini tidak berbeda dengan shutdown.
- awaitTermination
  - Blokir sampai semua task selesai. Harus shutdown() terlebih dahulu.

Threading Level Lebih Rendah

- Gunakan Thread.start(someRunnable)
  - Implementasikan Runnable, kirim ke konstruktor Thread, panggil start

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

  - Efek sama seperti taskList.execute(someRunnable), kecuali kita tidak dapat meletakkan loncatan pada jumlah thread simultan
- Perluas Thread
  - Letakkan metode run dalam sub-class Thread, instansiasi, panggil start

```
SomeThread t = new SomeThread(…);
t.start();
```

  - Warisan dari pre-Java-5; Kurang digunakan dalam aplikasi Java modern.

Topik Lanjutan

Callable

- Runnable
  - Metode “run” berjalan di background. Tidak ada nilai kembalian, tapi run dapat mengerjakan efek samping.
  - Gunakan “execute” untuk meletakkan dalam antrian tugas (*task queue*)
- Callable
  - Metode “call” berjalan di background. Mengembalikan suatu nilai yang dapat diretrieve setelah terminasi dengan “get”.
  - Gunakan “submit” untuk meletakkan dalam task queue.
  - Gunakan invokeAny dan invokeAll untuk memblokir sampai value atau values tersedia
    - Contoh: kita punya daftar link dari suatu halaman web dan ingin memeriksa statusnya (404 vs. good). Submit daftar link-link tersebut ke suatu task queue untuk berjalan secara konkuren, kemudian invokeAll akan memperlihatkan kepada kita nilai kembalian ketika semua link selesai diperiksa.

Jenis Antrian Task

- `Executors.newFixedThreadPool(nThreads)`
  - Jenis yang paling mudah dan luas digunakan. Membuat suatu list of tasks untuk dijalankan di latar belakang, tetapi dengan keberatan bahwa tidak pernah ada lebih dari nThread thread simultan yang berjalan.
- `Executors.newScheduledThreadPool`
  - Memungkinkan kita mendefinisikan task-task yang berjalan setelah suatu delay, atau yang berjalan secara berkala. Penggantian bagi kelas Timer pre-Java-5.
- `Executors.newCachedThreadPool`
  - Versi optimis untuk aplikasi yang berawal dengan banyak thread *short-running*. Menggunakan ulang instance thread.
- `Executors.newSingleThreadExecutor`
  - Membuat antrian task dan mengeksekusi satu demi satu (pada sau waktu)
- `ExecutorService (subclass) constructors`
  - Memungkinkan kita membangun antrian FIFO, LIFO dan prioritas.

Penghentian suatu Thread

```
public class SomeTask implements Runnable {
  private volatile boolean running;

  public void run(){
    running = true;
    while (running) {
      ...
    }
    doCleanup();
  }

  public void setRunning(boolean running) {
    this.running = running;
  }
}
```

Compiler pada sistem multiprocessor sering melakukan optimisasi yang mencegah perubahan variabel dari satu thread diketahui (terlihat) oleh thread lain. Untuk menjamin bahwa thread lain melihat perubahan itu, kita dapat gunakan metode **synchronized**, mendeklarasikan variabel "**volatile**", atau menggunakan **AtomicBoolean**.

Bug Sinkronisasi Jelek (Bagian 1)

```
public class Driver {
  public void startThreads() {
    …
    for(…) {
      taskList.execute(new SomeHandler(...));
    }
  }
}
```

Kelas sendiri atau inner class. Tetapi masalah ini tidak terjadi pada saat kita meletakkan “this” di sini.

Bug Sinkronisasi Jelek (Bagian 2)

```
public class SomeHandler implements Runnable {
  public synchronized void doSomeOperation() {
    accessSomeSharedObject();
  }
  ...
  public void run() {
    while(someCondition) {
      doSomeOperation();     // akses shared data
      doSomeOtherOperation();// bukan shared data
    }
  }
}
```

Kata kunci ini tidak punya efek apapun dalam konteks ini. Mengapa?

Solusi Sinkronisasi

- Solusi 1: Sinkronisasi pada *outer class*
  - Jika handler adalah suatu *inner class*, bukan kelas tersendiri

```
public OuterClassName {
  public void someMethod() {
    ...
    taskList.execute(new SomeHandler(...));
  }

  private class SomeHandler implements Runnable {
    public void run() { ... }

    public void doSomeOperation() {
      synchronized(OuterClassName.this) {
        accessSomeSharedObject();
      }
    }
  }
}
```

Solusi Sinkronisasi (Lanj.)

- Solusi 2: Sinkronisasi pada *shared data*

```
public void doSomeOperation() {
  synchronized(someSharedObject) {
    accessSomeSharedObject();
  }
}
```

- Solusi 3: Sinkronisasi pada obyek kelas

```
public void doSomeOperation() {
  synchronized(SomeHandler.class) {
    accessSomeSharedObject();
  }
}
```

Catatan:

- Jika kita menggunakan “synchronized” terhadap metode static, lock-nya adalah obyek kelas yang bersesuaian, bukan “this”.

Solusi Sinkronisasi (Lanj.)

- Solusi 4: Sinkronisasi pada obyek tertentu

```
public class SomeHandler implements Runnable{
  private static Object lockObject = new Object();
  ...
  public void doSomeOperation() {
    synchronized(lockObject) {
      accessSomeSharedObject();
    }
  }
  ...
}
```

- Mengapa masalah ini biasanya tidak terjadi dengan mekanisme thread kedua (dengan metode run di kelas main)?

Penentuan Ukuran Maksimum Thread Pool

- Pada banyak aplikasi, tebakan logis cukup membantu

```
int maxThreads = 100;
ExecutorService taskList =
  Executors.newFixedThreadPool(maxThreads);
```

Penentuan Ukuran Maksimum Thread Pool

- Jika kita perlu nilai lebih tepat

```
maxThreads = numCpus * targetUtilization * (1 +
  avgWaitTime/avgComputeTime)
```

  - Hitung numCpus dengan Runtime.getRuntime().availableProcessors()
  - targetUtilization adalah 0.0 s.d 1.0
  - Temukan rasio menunggu untuk menghitung waktu dengan *profiling*
  - Rumus tersebut di ambil dari buku *Java Concurrency in Practice*

Topik Lanjut Lain

- wait/waitForAll
  - Melepaskan kunci ke thread lain dan mensuspend dirinya (diletakkan di dalam antrian yang berasosiasi dengan kunci)
  - Sangat penting dalam beberapa aplikasi, tetapi sangat sulit menjaga ketepatan.
- notify/notifyAll
  - Membangunkan semua thread yang menunggu kunci
  - Thread yang diberitahu tidak langsung mulai eksekusi, tetapi diletakkan di dalam antrian thread runnable
- Utilitas konkurensi dalam java.util.concurrency
  - Utilitas threading lanjutan termasuk semaphores, koleksi yang didesain untuk aplikasi berthread banyak, operasi atomik, dll.
- Mendebug masalah thread
  - Gunakan JConsole
    - http://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html

Pertanyaan?

Rangkuman

- Pendekatan dasar

```
ExecutorService taskList =
        Executors.newFixedThreadPool(poolSize);
```

- Tiga variasi
  - `taskList.execute(new SeparateClass(…));`
  - `taskList.execute(this);`
  - `taskList.execute(new InnerClass(…));`
- Penanganan shared data

```
synchronized(referenceSharedByThreads) {
    getSharedData();
    modifySharedData();
}
doOtherStuff();
```

Sebenarnya 4 variasi, dengan lambdas adalah yang ke-4-nya. Tetapi lambda expressions tidak dicakup di kuliah ini.

Referensi

- Referensi online
  - Tutorial Java 8
    - http://www.coreservlets.com/java-8-tutorial/
  - Lesson: Concurrency (Oracle Java Tutorial)
    - http://docs.oracle.com/javase/tutorial/essential/concurrency/
  - Jacob Jenkov's Concurrency Tutorial
    - http://tutorials.jenkov.com/java-concurrency/index.html
  - Lars Vogel's Concurrency Tutorial
    - http://www.vogella.de/articles/JavaConcurrency/article.html