Berner Fachhochschule - Technik und Informatik

Object-Oriented Programming 2

Topic 4: Multithreading

Prof. Rolf Haenni & Prof. Annett Laube

FS 2016

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Outline

Outline

Motivation

- A thread is a program unit that is executed independently of other parts of the program
- Motivation for threads:
  - → Different components of a system run at different speed
  - → Examples: user (5 keystrokes/s.), harddisk (200 accesses/s.), processor ($2 * 10^9$ instructions/s.)
- In a single threaded application, the processor is completely blocked while waiting for the responses of other (slower) system components
- Multi-threaded applications make better use of the available resources and are more responsive

Thread Scheduler

- The JVM executes each thread alternately for a short amount of time (called time slice), which gives the impression of parallel execution
- The thread scheduler is the JVM component that distributes the available computational resources to the threads
- There is no guarantee about the order in which threads are executed or about the length of the time slices
- The actual running time of an algorithm running in a thread depends on the amount and the types of other threads that are currently running
- On a multi-core system, multiple threads can be executed in parallel, with every core executing one or multiple threads separately

Threads vs. Processes

- Threads differ from traditional processes in multitasking operating system
  - → Multiple threads belong to a single application, while processes are typically independent
  - → Threads share memory (and other resources), while multiple processes use separated memory areas
  - → Threads share a common address space, while processes have separate address spaces
  - → Threads can communicate directly, while processes interact through system-provided inter-process communication mechanisms
  - → Switching between threads is typically faster than switching between processes

Outline

Running a Thread: Method 1

- Code to be executed in a thread can be contained within the method `void run()` defined by the interface `Runnable`

```
public class MyRunnable implements Runnable {

  @Override
  public void run() {
    // Write your code to be executed in a thread here
  }

}
```

- Objects of type `Runnable` must then be added to a thread, which executes the code when the thread is started

Running a Thread: Method 1

```java
public class HelloWorld implements Runnable {

  @Override
  public void run() {
    try {
      Thread.sleep(5000);
      System.out.println("Hello World");
    } catch (InterruptedException e) {}
  }

  public static void main(String[] args) {
    Runnable helloWorld = new HelloWorld();
    Thread thread = new Thread(helloWorld);
    System.out.println("Starting...");
    thread.start();
  }
}
```

Running a Thread: Method 1

```
public class HelloWorldAnonymous {

  public static void main(String[] args) {
    Runnable helloWorld = new Runnable(){
      @Override
      public void run() {
        try {
          Thread.sleep(5000);
          System.out.println("Hello World");
        } catch (InterruptedException e) {}
      }
    };
    Thread thread = new Thread(helloWorld);
    System.out.println("Starting...");
    thread.start();
  }
}
```

Running a Thread: Method 1

```java
public class HelloWorldLambda {

  public static void main(String[] args) {
    Thread thread = new Thread(() -> {
      try {
        Thread.sleep(5000);
        System.out.println("Hello World");
      } catch (InterruptedException e) {}
    });
    System.out.println("Starting...");
    thread.start();
  }
}
```

Running a Thread: Method 2

- The second option of defining a thread consists in creating a subclass of `Thread`
- Since `Thread` itself implements `Runnable`, the `run()` method can be overridden
- The thread's own `run()` method is only called if the thread is constructed without giving an object of type `Runnable`
- The default `run()` method of a thread does nothing

Running a Thread: Method 2

```
public class HelloWorldThread extends Thread {

  @Override
  public void run() {
    try {
      Thread.sleep(5000);
      System.out.println("Hello World");
    } catch (InterruptedException e) {}
  }

  public static void main(String[] args) {
    Thread thread = new HelloWorldThread();
    System.out.println("Starting...");
    thread.start();
  }
}
```

Example with Multiple Threads I

```
public class HelloAnyone implements Runnable {

  private String name;
  private int delay;
  private int n;

  public HelloAnyone(String name, int delay, int n) {
    this.name = name;
    this.delay = delay;
    this.n = n;
  }

  @Override
  public void run() {
    for (int i = 0; i < this.n; i++) {
      try {
        Thread.sleep(this.delay);
```

Example with Multiple Threads II

```
        System.out.println("Hello " + this.name);
      } catch (InterruptedException e) {}
    }
  }

  public static void main(String[] args) {
    Thread thread1 = new Thread(new HelloAnyone("World", 3000,
         5));
    Thread thread2 = new Thread(new HelloAnyone("Universe",
        1000, 10));
    System.out.println("Starting...");
    thread1.start();
    thread2.start();
  }
}
```

Terminating Threads

- A thread terminates when its `run()` method terminates
- Do not use the deprecated `stop()` method, instead notify a thread that it should terminate:

  → `thread.interrupt();`

- Note that calling `interrupt()` does not terminate a thread, it only sets a Boolean variable called interrupt status
- Any loop within `run()` method should check occasionally whether it has been interrupted:

  → `while (!Thread.interrupted()) {...};`

- If a thread is blocked on a `Thread.sleep()`, it is unblocked by receiving an `InterruptedException` and its interrupt status is cleared

Terminating Threads: Example I

```
public class HelloWorldInterruped {

  public static void main(String[] args) {

    Thread thread = new Thread(() -> {
      try {
        while (!Thread.interrupted()) {
          Thread.sleep(500);
          System.out.println("Hello World");
        }
        System.out.println("Loop interrupted");
      } catch (InterruptedException e) {
        System.out.println("Sleeping interrupted");
      }
    });

    System.out.println("Starting...");
```

Terminating Threads: Example II

```
    thread.start();
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {}
    thread.interrupt();
  }
}
```

Thread Pools

- Starting and terminating threads is relatively expensive
- In case of a large number of small tasks, it is not recommended executing them by individual threads
- A thread pool consists of a n of permanent threads created to perform m tasks concurrently
  - → Typically n is not equal to m
  - → n is tuned to the computing resources available (processors, cores, memory)
  - → m depends on the problem and may not be known upfront
- The thread pool executer allocates the tasks to the n threads, which may imply that some tasks need to be queued

Thread Pools in Java

- In Java, thread pools are represented by the the interfaces
  - → `ExecutorService`, `ScheduledExecutorService`

  and the classes
  - → `ThreadPoolExecutor`, `ScheduledThreadPoolExecutor`
- Creating thread pools is simplified by some static methods in the helper class `Executors`
  - → `ExecutorService newSingleThreadExecutor()`
  - → `ExecutorService newFixedThreadPool(int n)`
  - → `ScheduledExecutorService newSingleThreadScheduledExecutor()`
  - → `ScheduledExecutorService newScheduledThreadPool(int n)`

The ExecutorService Interface

- Methods of the interface `ExecutorService`:
  - → `execute(Runnable task)`
  - → `shutdown()`
  - → `isShutdown()`
  - → `isTerminated()`
  - → `awaitTermination(long timeout, TimeUnit u)`
- Additional methods of `ScheduledExecutorService`:
  - → `schedule(Runnable task, long delay, TimeUnit u)`
  - → `scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit u)`
  - → `scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit u)`

Thread Pools: Example

```
public class Task implements Runnable {
  private int id;
  public Task(int id) {
    this.id = id;
  }

  @Override
  public void run() {
    System.out.println(Thread.currentThread().getName() + ":
         start task " + this.id);
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {}
    System.out.println(Thread.currentThread().getName() + ":
         end task " + this.id);
  }
}
```

Thread Pools: Example

```
public class ThreadPool {

  public static void main(String[] args) {

    ExecutorService executor = Executors.newFixedThreadPool(5)
        ;
    for (int i = 0; i < 12; i++) {
      Runnable task = new Task(i);
      executor.execute(task);
    }
    executor.shutdown();
    do { // nothing
    } while (!executor.isTerminated());
    System.out.println("All threads finished");
  }
}
```

Thread Pools: Example I

```
public class ScheduledThreadPool {

  public static void main(String[] args) {

    ScheduledExecutorService executor = Executors.
        newScheduledThreadPool(1);

    executor.scheduleAtFixedRate(() -> {
        System.out.println("start new task");
      }, 3, 1, TimeUnit.SECONDS);

    executor.schedule(() -> {
      System.out.println("Hello World");
    }, 5, TimeUnit.SECONDS);

    executor.schedule(() -> {
        executor.shutdown();
```

Thread Pools: Example II

```
    }, 10, TimeUnit.SECONDS);

    do { // nothing
    } while (!executor.isTerminated());
    System.out.println("All threads finished");
  }
}
```

Outline

Introduction

Working With Threads

Race Conditions

Deadlocks

Synchronized Collections

Race Conditions

- In thread programming, it can happen that a thread's time slice ends when an object is in an inconsistent state
- If another thread continues to work on the same object, we can get unexpected errors
- This kind of problem is called race condition (this term originates with the idea of two signals racing each other to determine the output first)
- Race conditions are difficult to reproduce and debug, since the end result is nondeterministic and depends on the relative timing between interfering threads

Race Conditions

Thread 1	Thread 2	Value
		0
read		0
increase		0
write		1
	read	1
	increase	1
	write	2

Thread 1	Thread 2	Value
		0
read		0
	read	0
increase		0
write		1
	increase	1
	write	1

Race Conditions: Example I

```
public class Counter {

  private int value = 0;

  public void countUp() {
    this.value++;
  }

  public void countDown() {
    this.value--;
  }

  public static void main(String[] args) {

    int rounds = 100000;
    Counter counter = new Counter();
```

Race Conditions: Example II

```
    Thread thread1 = new Thread(() -> {
      for (int i = 0; i < rounds; i++)
        counter.countUp();
    });
    thread1.start();

    Thread thread2 = new Thread(() -> {
      for (int i = 0; i < rounds; i++)
        counter.countDown();
    });
    thread2.start();

    do { // nothing
    } while (thread1.isAlive() || thread2.isAlive());
    System.out.println(counter.value);
  }
}
```

Synchronized Methods or Blocks

- The simplest method to avoid race conditions is to declare synchronized methods
  - → `public void synchronized countUp() {...}`
  - → `public void synchronized countDown() {...}`

  (for this to work, each Java object offers an intrinsic lock)
- Another solution is to declare synchronized blocks of code
  - → Using this object's intrinsic lock

    ```
    synchronize(this) {this.value++;}
    synchronize(this) {this.value--;}
    ```
  - → Using some other object's intrinsic lock

    ```
    Object lock = new Object();
    synchronize(lock) {this.value++;}
    synchronize(lock) {this.value--;}
    ```

Synchronized Methods or Blocks: Example I

```
public class SynchronizedCounter {

  private int value = 0;

  public synchronized void countUp() {
    this.value++;
  }
  public void countDown() {
    synchronized (this) {
      this.value--;
    }
  }

  public static void main(String[] args) {

    int rounds = 100000;
    SynchronizedCounter counter = new SynchronizedCounter();
```

Synchronized Methods or Blocks: Example II

```
    Thread thread1 = new Thread(() -> {
      for (int i = 0; i < rounds; i++)
        counter.countUp();
    });
    thread1.start();

    Thread thread2 = new Thread(() -> {
      for (int i = 0; i < rounds; i++)
        counter.countDown();
    });
    thread2.start();

    do { // nothing
    } while (thread1.isAlive() || thread2.isAlive());
    System.out.println(counter.value);
  }
}
```

Avoiding Race Conditions With Locks

- The most flexible way to avoid race conditions is to explicitly protect critical code with a lock
- Java provides an interface `Lock` and a class `ReentrantLock` with methods
  - → `lock()`
  - → `unlock()`
- If a thread calls `lock()` for an unlocked lock, the thread owns the lock until it calls `unlock()`
- If a thread calls `lock()` for an locked lock, the thread is temporarily deactivated
- A deactivated thread is reactivated periodically so that it can try again to acquire the lock

Example of Using Locks I

```
public class CounterWithLock {

  private int value = 0;
  private Lock lock = new ReentrantLock();

  public void countUp() {
    lock.lock();
    this.value++;
    lock.unlock();
  }

  public void countDown() {
    lock.lock();
    this.value--;
    lock.unlock();
  }
```

Example of Using Locks II

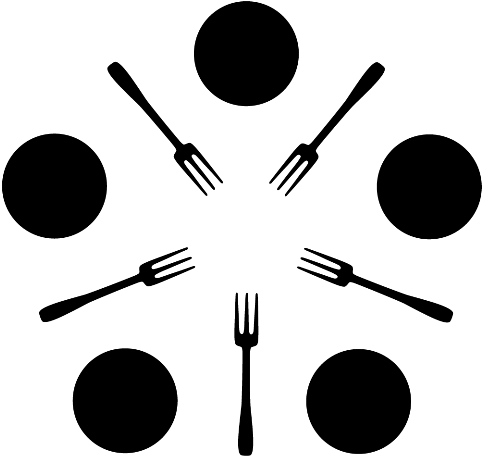
```
public static void main(String[] args) {

  int rounds = 100000;
  CounterWithLock counter = new CounterWithLock();

  Thread thread1 = new Thread(() -> {
    for (int i = 0; i < rounds; i++)
      counter.countUp();
  });
  thread1.start();

  Thread thread2 = new Thread(() -> {
    for (int i = 0; i < rounds; i++)
      counter.countDown();
  });
  thread2.start();

  do { // nothing
```

Example of Using Locks III

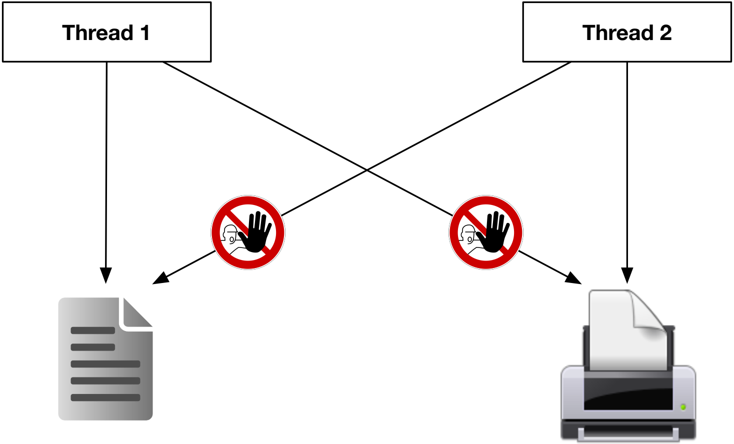
```
    } while (thread1.isAlive() || thread2.isAlive());
    System.out.println(counter.value);
  }
}
```

Outline

Dining Philosophers



Typical Deadlock



Typical Deadlock: Example I

```
public class FilePrinter {

  private static Object fileLock = new Object();
  private static Object printerLock = new Object();
  private static int n = 30000;

  public static void main(String[] args) {

    new Thread(() -> {
      synchronized (fileLock) {
        for (int i = 0; i < n; i++) {} // do something
        synchronized (printerLock) {
          System.out.println("Task 1: printing file");
        }
      }
    }).start();
```

Typical Deadlock: Example II

```
    new Thread(() -> {
      synchronized (printerLock) {
        for (int i = 0; i < n; i++) {} // do something
        synchronized (fileLock) {
          System.out.println("Task 2: printing file");
        }
      }
    }).start();
  }
}
```

Deadlocks

- Deadlocks can occur only if ...
  - → two or more threads are using the same locks, and
  - → two or more locks are in use
- The simplest strategies to avoid deadlock are:
  - → Never use more than one lock at a time
  - → Lock multiple locks always in the same order
  - → Lock all locks at once or none at all
  - → Queue all tasks in one single thread and synchronize the queue (e.g. using `Platform.runLater(Runnable task)` for queuing GUI tasks in the JavaFX application thread)

Typical Deadlock: Solution I

```
public class FilePrinterOrderedLocks {

  private static Object fileLock = new Object();
  private static Object printerLock = new Object();
  private static int n = 100;

  public static void main(String[] args) {

    new Thread(() -> {
      synchronized (fileLock) {
        for (int i = 0; i < n; i++) {} // do something
        synchronized (printerLock) {
          System.out.println("Task 1: printing file");
        }
      }
    }).start();
```

Typical Deadlock: Solution II

```
    new Thread(() -> {
      synchronized (fileLock) {
        for (int i = 0; i < n; i++) {} // do something
        synchronized (printerLock) {
          System.out.println("Task 2: printing file");
        }
      }
    }).start();
  }
}
```

Outline

Synchronized Collections

- Java collection such as lists, sets, or maps are not thread-safe, i.e., race conditions may appear when used in multiple threads
- The helper class Collections provides several static methods to construct thread-safe wrapper classes:
  - → Collection<T> synchronizedCollection(Collection<T> collection)
  - → List<T> synchronizedList(List<T> list)
  - → Set<T> synchronizedSet(Set<T> set)
  - → SortedSet<T> synchronizedSortedSet(SortedSet<T> set)
  - → Map<K,V> synchronizedMap(Map<K,V> map)
  - → SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> map)

Synchronized Collections I

```
public class SynchronizedLists {

  public static void main(String[] args) throws Exception {

    List<Integer> list1 = new ArrayList<>();

    new Thread(() -> {
      for (int i=0; i<1000; i++) {
        list1.add(i);
      }
    }).start();

    new Thread(() -> {
      for (int i=0; i<1000; i++) {
        list1.add(i);
      }
    }).start();
```

Synchronized Collections II

```
List<Integer> list2 = Collections.synchronizedList(new
    ArrayList<>());

new Thread(() -> {
  for (int i=0; i<1000; i++) {
    list2.add(i);
  }
}).start();

new Thread(() -> {
  for (int i=0; i<1000; i++) {
    list2.add(i);
  }
}).start();

Thread.sleep(100); // wait for threads to terminate
System.out.println(list1.size());
```

Synchronized Collections III

```
    System.out.println(list2.size());
  }

}
```