Concurrent Programming

Vadim Zaytsev
Programming, FB4,
Universität Koblenz-Landau

19 June 2008

Concurrent programming

- Parallel computations — multiple CPUs
- Distributed systems — physical separation
- Multi-threaded programs — Java way
- Asynchronous execution — arbitrarily sequenced
- Multitasking — what OS does
- Communicating processes — theory behind it

Motivation

- Responsive UI
- Throughput optimisation
- Real multitasking/parallellism
- Programming convenience

Dining philosophers

- Edsger Dijkstra and Tony Hoare, 1971
- Five philosophers are sitting at a round table with a bowl of spaghetti in the centre.
- They eat or think. While eating, they are not thinking, and while thinking, they are not eating.
- To eat, each one needs two forks.
- Five forks total, one between any adjacent two.

Philosopher in Java

```
public class Philosopher implements Runnable
{
 public void run()
 {
  try{while(!Thread.interrupted())
  { pause();       // thinking
    right.take();
    left.take();
    pause();       // eating
    right.drop();
    left.drop();
  }}
  catch (InterruptedException e)
   {System.out.println(this+"killed.");}
}}
```

Fork in Java

```
public class Fork
{
  protected boolean taken = false;

  public synchronized void take() throws InterruptedException
  {
    while(taken)
      wait();
    taken = true;
  }
  public synchronized void drop()
  {
    taken = false;
    notifyAll();
  }
}
```

Threads in Java

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

public class Test
{public static void main(String[] args) throws Exception {
    int size = 5;
    Thread [] table = new Thread [5];
    Fork [] forks = new Fork [size];
    for(int i = 0; i < size; i++)
     forks[i] = new Fork();
    for(int i = 0; i < size; i++)
    { table[i] = new Thread(
            new Philosopher(i, forks[i], forks[(i+1) % size]));
      table[i].start();  }
    System.out.println("Press 'Enter' to quit");
    System.in.read();
    for(int i = 0; i < size; i++)
        table[i].interrupt();
}}
```

Thread pools in Java

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

public class Test
{
   public static void main(String[] args) throws Exception {
    int size = 5;
    ExecutorService exec = Executors.newCachedThreadPool();
    Fork[] forks = new Fork[size];
    for(int i = 0; i < size; i++)
      forks[i] = new Fork();
    for(int i = 0; i < size; i++)
      exec.execute(
          new Philosopher(i, forks[i], forks[(i+1) % size]));
    System.out.println("Press 'Enter' to quit");
    System.in.read();
    exec.shutdownNow();
  }
}
```

Starting a thread in Java

- A class extends **Thread** and its `start()` is called
- A class starts itself from constructor
- A class implements **Runnable** and is started after initialisation
- An object is executed through a pool
- ...

Finishing a thread in Java

- A thread finishes by itself
- `isAlive()` is explicitly checked
- `join()` is called on a thread
- An object is shut down through a pool
- ...

Dining philosophers — Problems

- **Starvation**: no guarantee for getting food
  (a thread perpetually denied resources)

- **Deadlock**: everyone picks up one fork
  (no progress while threads are waiting for one another)

Deadlock explained

- Shared reusable mutually exclusive resources
- Incremental acquisition of resources
- No pre-emption (confiscation)
- Idle waiting — no useful actions

Solution: timeout

- If a philosopher cannot get two forks in five minutes, he gives up, drops everything and goes back to thinking

DEMO

- Mind the timeout implementation

Dining philosophers v.2 — Problems

- **Starvation**: no guarantee for getting food
  (a thread perpetually denied resources)

- **Race hazard**: output is dependent on timing
  (different execution sequences yield different results)

- **Livelock**: only picking up and dropping forks
  (threads are working, but still no progress)

More on races and mutexes

- `synchronized` methods — monitors/semaphores
- `synchronized{}` code
- `java.util.concurrent.Semaphore` — counting
- **Barrier** — the opposite of monitor
- **Channel** — alternative concurrency model

Solution: left-handed philosopher

```
public class Test
{
  public static void main(String[] args) throws Exception {
    int size = 5;
    ExecutorService exec = Executors.newCachedThreadPool();
    Fork[] forks = new Fork[size];
    for(int i = 0; i < size; i++)
      forks[i] = new Fork();
    exec.execute(
          new Philosopher(0, forks[size-1], forks[0]));
    for(int i = 1; i < size; i++)
      exec.execute(
          new Philosopher(i, forks[i], forks[(i+1) % size]));
    System.out.println("Press 'Enter' to quit");
    System.in.read();
    exec.shutdownNow();
  }
}
```

Generalised solution

- Stabilising philosopher: one thread is working toward stability
- Partial ordering: any hierarchy of resources is good, if it breaks the circle
- Can be generalised even further

Closure on dining philosophers

- Several solutions introduce new entities: a book, a waiter, an activity, a communication channel, a resource state, etc — all map to real life problems

- Starving philosophers — a generalisation for demonstrating implementation problems for wait/notify mechanism

- Many sequels and spin-offs: dining cryptographers, drinking philosophers, mobile philosophers, triple-handed dining philosophers, evolving philosophers, congenial talking philosophers, etc

Java thread features, recollected

- Thread and Runnable
- start(), interrupt(), join()
- wait(), notify(), notifyAll()
- getPriority(), setPriority()
- synchronized, volatile
- thread pools

Legacy Java

- What was wrong with `Thread.stop()`?
  - What if `Thread.interrupt()` does not work?
  - What if we catch `ThreadDeath` exception?
- What was wrong with `Thread.suspend()` and `Thread.resume()`?
- Monitors are there for a reason!

Futures/promises

```
ExecutorService executor = Executors.newFixedThreadPool(1);
FutureTask<String> future =
 new FutureTask<String>(
   new Callable<String>()
    {
     public String call()
      {
        return doSomeSlowStuff();
      }
    });
executor.execute(future);
```

Thread safety

- Preserving correct behaviour when being executed simultaneously by multiple threads.
- Two ways to implement: re-entrancy and mutual exclusion
- Atomicity/linearisability: all or nothing at all

List of example packages

- `dining.simple` — deadlock-prone Dining Philosophers
- `dining.threadpool` — the same, with a threadpool
- `dining.timeout` — deadlock-free, livelock-prone, busy loop
- `dining.stabilised` — left-handed philosopher
- `futures` — string reversion with Futures
- `nonthreadsafe` — error-prone credit card transactions

The End.