

Distributed Systems

4. Programming with Threads

Werner Nutt

1

Processes vs. Threads

Components of distributed systems have to do different things at the same time (concurrency)

Realization by processes is expensive

- processes have separate resources (e.g., memory space)
- ➔ switching between processes keeps the kernel busy

Threads are cheaper

- run in one process (each process has at least one)
- share memory space and other resources
(which gives rise to other difficulties)

2

Threads in Java

- Threads are first class objects
 - instances of the class **Thread** —
or of a subclass of **Thread**, created by the programmer
- In every program, there is a thread “main”
- The class **Thread** implements the interface **Runnable**
 - Runnable has only one method: **run()**
- Threads can be constructed from implementations of **Runnable**, using constructors, like
 - **new Thread(Runnable target)**
 - **new Thread(Runnable target, String name)**

3

Threads in Java (cntd)

- Threads have the following methods
 - **run()**
 - **start()**
 - **getPriority()**
 - **join()** *(waits for the thread to die)*
- The class **Thread** has also static methods, e.g.,
 - **yield()** *(lets the current thread pause)*
 - **sleep(long n)** *(lets the current thread pause
for n milliseconds)*
- Threads can be daemons and may belong to groups

4

Extending the Class Thread (Example)

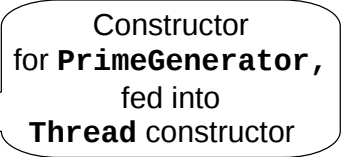
```
class PrimeThread extends Thread {  
    long minPrime;  
  
    PrimeThread(long minPrime) {  
        this.minPrime = minPrime;  
    }  
  
    public void run() {  
        // compute primes larger than minPrime  
        . . .  
    }  
}
```



5

Constructing a Thread from a “Runnable”

```
class PrimeGenerator implements Runnable {  
    long minPrime;  
    PrimeGenerator(long minPrime) {  
        this.minPrime = minPrime;  
    }  
  
    public void run() {  
        // compute primes larger than minPrime  
        . . .  
    }  
}
```



```
Thread primeThread =  
    new Thread(new PrimeGenerator(1000));  
  
primeThread.start();
```

Is this better? If so, why?

6

Putting a Thread Asleep

Static methods

Thread.sleep(ms)

Thread.sleep(ms, ns)

- Current thread pauses for (approx.) the indicated time
- Useful for
 - making processor time available for other threads
 - ensuring that thread proceeds with a defined rhythm

("pacing" a thread)

7

Thread Interference

```
public class Counter {  
    int c = 0;  
  
    public void increment() { c++; }  
    public void decrement() { c--; }  
    public int value() { return c; }  
}
```

- In reality, **increment()** and **decrement()** are complex operations
- Two threads A, B may interfere when accessing the same counter
- Aim: B must see the effect of A's action (or vice versa)
 - ➔ A "happens before" B

8

Synchronisation

Achieves “happens before” relationship
between threads accessing an object

Principles:

- Every object has an intrinsic lock (= “monitor”)
- A lock for on object is acquired e.g., by executing
 - a synchronised method of that object, or
 - a synchronized statement for that object (*see below*)
- Methods can be declared as synchronized
 - e.g., **public synchronized void updateBalance(..)** for class **Account**
 - **acc.updateBalance(...)** can only be executed by a thread if the thread has a lock for **acc**
 - when the method call is completed, the lock is released

9

Example: A Synchronized Counter

```
public class SynchronizedCounter {  
    int c = 0;  
  
    public synchronized void increment() {  
        c++;  
    }  
  
    public synchronized void decrement() {  
        c--;  
    }  
  
    public synchronized int value() {  
        return c;  
    }  
}
```

10

Synchronization Wrappers for Collections

Collections (Set, List, Map, SortedSet, and SortedMap)

are typical data structures to be shared by several threads

➔ need for synchronization

- Factory methods of class Collections can make a collection object "thread safe"

```
List msgQueue =  
    Collections.synchronizedList(new LinkedList());
```

- Iteration has to be synchronized by a synchronized statement

```
synchronized(msgQueue) {  
    Iterator i = msgQueue.iterator();  
        // must be in synchronized block  
    while (i.hasNext())  
        send(i.next());  
}
```

11

Deadlocks

Scenario: two threads T_1 , T_2

- T_1 has a lock for object O_1 , T_2 has a lock for object O_2
- T_1 needs a lock for O_2 to complete work on O_1 ,
 T_2 needs a lock for O_1 to complete work on O_2

➔ Deadlock

12

References

In preparing the lectures I have used several sources.
The main one is the following:

Web:

- The Java tutorials, Lesson Concurrency

<http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html>