Programmierpraktikum

Claudius Gros, SS2012

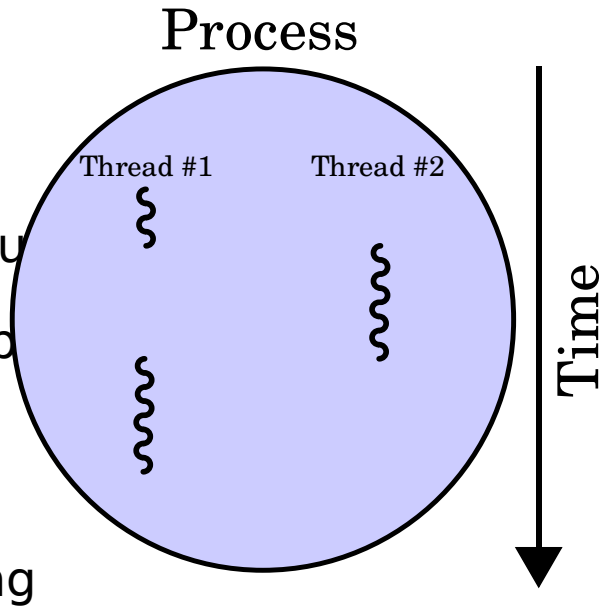
Institut für theoretische Physik
Goethe-University Frankfurt a.M.

Processes and Threads

processes vs. threads

processes

- everything happing in a computer is a process
- **time slicing**
  distribution of time allocated to all processes ru
- `top`: Linux console command listing all active p



threads

- every process has at least one thread executing
- **multi-threaded execution**
  within one process (program), execution is carried out asynchronously (independently) by several threads
- the process (program) continues as long as there is at least one thread still running

parallelization

- distribution of computing load onto several (many)
  CPUs (central processing unit) or GPUs (graphics processing unit)
- **message passing interface (MPI)**
  standardized message-passing system
  for parallelization (Fortran, C++, ...)
- Java 7 Fork and Join framework
  $\rightarrow$ advanced numerical computing

sleeping

- `main()` is called by an a priori `Thread`
- `System.currentTimeMillis()` returns the current time

```
 1. import java.util.*;
 2.
 3. public class SleepingDemo {
 4.   public static void main(String args[])
 5.         throws InterruptedException            // needed when working with threads
 6.   {
 7.   long startTime = System.currentTimeMillis();  // current time in milli seconds
 8.   for (int i = 0; i< 5; i++)
 9.     {
10.     Thread.sleep(2000);                        // pause current thread for 2 seconds
11.     System.out.printf("time since start: %5d ms\n",
12.                       System.currentTimeMillis()-startTime);
13.     }
14.   }  // end of  main()
15. }    // end of SleepingDemo
```

runnable classes

- an instantiated `class` implementing *runnable*
  can have its own thread
- any runnable class needs to habe a member function `.run()`
  - *`.run()` is executed when the thread is started*
  - *the thread stops when reaching the end of `.run()`*
  - *as for the `.main()` method*
- one can *interrrupt* a thread or
  join a thread (waiting for the thread to die)

threads and interrupts

- an *interrupt* is a notification, not a termination signal
- threads stop at the end of `.run()`
- Java is platform independent, use the code to test the performace of various CPUs

```
 1. import java.util.*;
 2.
 3. public class ThreadDemo {
 4.   public static void main(String args[])
 5.         throws InterruptedException           // needed when working with threads
 6.   {
 7.   long startTime = System.currentTimeMillis();
 8.
 9.   SomeOperations some = new SomeOperations(); // class instantiation
10.   Thread t = new Thread(some);                // create new thread
11.   t.start();                                  // start some.run()
12.
```

```
13.   for (int i = 0; i < 6; i++)
14.     {
15.     Thread.sleep(2000);                       // pause thread of main()
16.     System.out.printf("   time since start: %5d ms\n",
17.             System.currentTimeMillis()-startTime);
18.     System.out.printf("nOperations/1000000: %5d\n",
19.                   some.nOperations/1000000);
20.
21.     if (i==3)                                 // thread would run ad infinitum
22.       t.interrupt();                          // if not interupted
23.     }
24.
25.   }  // end of ThreadDemo.main()
26. }    // end of class ThreadDemo
27.
28. // *** **************
29. // *** runnable class
30. // *** **************
31.
32. class SomeOperations implements Runnable {
33.   double number = 1.0;
34.   public long nOperations = 0;
35.
```

```
36.   public void run()              // every runnable class needs
37.   {                              // this member function
38.   long temp;
39.   while (!Thread.interrupted())  // infinite loop!
40.     {
41.     nOperations++;
42.     number = number * (1.0+1.0/nOperations);
43.     }
44.   System.out.printf("\n %s \n\n","I have been interrupted, stopping now");
45.   }  // end of SomeOperations.run()
46. }    // end of class SomeOperations
```

exercise: threads

large number of threads on a single processor

Write a program generating a large number of threads and measuring the resulting computational performance.

- the *runnable* `class` does some simple caculations if not *interrupted*
- instantiate $N = 1, \ldots, 1000$ copies of the runnable class and start their threads
- measure the overal performace, the total number of computations performed, as a function of the number of threads N

the Java virtual maschine (JVM)

platform independent computing

- a component of the Java runtime environment (JRE)
- executes platform-independent bytecodes (*.class* files)
- a JIT (just in time) compiler translates the bytecode into the native machine code (operating system calls)
- other components of the JRE are
  - *heap memory (memory allocated for the program)*
  - *garbage collection*
  - *...*

garbage collection (GC)

- automatic memory management

- reclaiming memory occupied by objects no longer used
- automatic GC in Java, no memory leaks
- (partly) manual in C, C++
-