G52CON: Concepts of Concurrency

Lecture 1: Introduction

Natasha Alechina
School of Computer Science
nza@cs.nott.ac.uk

Outline of this lecture

- what is concurrency
- sequential vs concurrent programs
- applications of concurrency
- module aims and objectives
- scope of the module
- assessment
- suggested reading

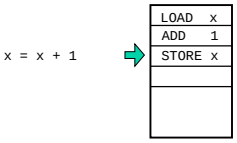
What is concurrency

Concurrent programs are programs where several processes are executing simultaneously:

- operating systems
- servers
- database management systems
- GUI applications (for example where one thread is painting images and another is responding to user input)

Sequential programs

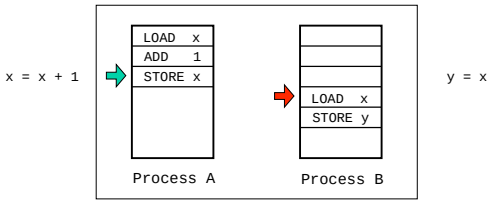
All programs are sequential in that they execute a sequence of instructions in a pre-defined order:



There is a *single thread* of execution or control.

Concurrent programs

A **concurrent program** is one consisting of two or more processes — threads of execution or control



Each *process* is itself a sequential program.

Example (from Arnold and Gosling)

Create a class PingPong, subclass of a Thread (in the next lecture) which every **`delay`** milliseconds prints a String **`word`**. **`delay`** and **`word`** are passed to the PingPong's constructor.

```
public class PingPong extends Thread {
  private String word; // what word to print
  private int delay; // how long to pause
  public PingPong(String whatToSay, int
                                  delayTime) {
   word = whatToSay;
   delay = delayTime;
}
```

Example continued: run method

```
public void run() {
  try {
    while(true) {
      System.out.print(word + " ");
      sleep(delay); // wait until next time
    }
  } catch (InterruptedException e) {
    return; // end this thread
  }
}
```


Example continued: main

```
public static void main(String[] args) {
  new PingPong("ping", 33).start();
  new PingPong("PONG", 100).start(); }
```

This main method creates two instances of the class. One prints "ping" every 33 ms and another prints "pong" every 100 ms.

If you run this you get something like this on the screen:

`ping PONG ping ping PONG ping ping ping PONG ping ping PONG …`

which is totally useless, as such, but shows how to spawn two processes in one program.

Proper example: file downloading

Consider a client–server system for file downloads (e.g. BitTorrent, FTP)

- without concurrency
  - it is impossible to interact with the *client* (e.g., to cancel the download or start another one) while the download is in progress
  - the *server* can only handle one download at a time—anyone else who requests a file has to wait until your download is finished
- with concurrency
  - the user can interact with the client while a download is in progress (e.g., to cancel it, or start another download)
  - the server can handle multiple clients at the same time

In parallel on a single processor?

- When several processes are executed on a single processor clearly they are not running in *true parallel*; instead they are executed in an *interleaved* fashion.
- Concurrent designs of a program can still be effective even if you only have a single processor:
  - many sequential programs spend considerable time blocked, e.g. waiting for memory or I/O
  - this time can be used by another thread in your program (rather than being given by the OS to someone else's program)
  - it's also more portable, if you do get another processor

More examples of concurrency

- **GUI-based applications:** e.g., `javax.swing`
- **Mobile code:** e.g., `java.applet`
- **Web services:** HTTP daemons, servlet engines, application servers
- **Component-based software:** Java beans often use threads internally
- **I/O processing:** concurrent programs can use time which would otherwise be wasted waiting for slow I/O
- **Real Time systems:** operating systems, transaction processing systems, industrial process control, embedded systems etc.
- **Parallel processing:** simulation of physical and biological systems, graphics, economic forecasting etc.

Implementations of concurrency

Two main types of implementations of concurrency:

- **shared memory**: the execution of concurrent processes by running them on one or more processors all of which access a shared memory —processes communicate by reading and writing shared memory locations;
- **distributed processing**: the execution of concurrent processes by running them on separate processors—processes communicate by message passing.

Shared memory implementations

We can further distinguish between:

- **multiprogramming**: the execution of concurrent processes by timesharing them on a single processor (concurrency is *simulated*);
- **multiprocessing**: the execution of concurrent processes by running them on separate processors which all access a shared memory (*true parallelism* as in distributed processing).

… it is often convenient to ignore this distinction when considering shared memory implementations.

Cooperating concurrent processes

The concurrent processes which constitute a concurrent program must *cooperate* with each other:

- for example, downloading a file in a web browser generally creates a new process to handle the download
- while the file is downloading you can also continue to scroll the current page, or start another download, as this is managed by a different process
- if the two processes *don't* cooperate effectively, e.g., when updating the display, the user may see only the progress bar updates or only the updates to the main page.

Synchronising concurrent processes

To cooperate, the processes in a concurrent program must *communicate* with each other:

- communication can be programmed using *shared variables* or *message passing*;
  - when shared variables are used, one process *writes* into a shared variable that is *read* by another;
  - when message passing is used, one process *sends* a message that is *received* by another;
- ***the main problem in concurrent programming is synchronising this communication***

Competing processes

Similar problems occur with *functionally independent* processes which don't cooperate, for example, separate programs on a time-shared computer:

- such programs implicitly *compete* for resources;
- they still need to synchronise their actions, e.g., two programs can't use the same printer at the same time or write to the same file at the same time.

In this case, synchronisation is handled by the OS, using similar techniques to those found in concurrent programs.

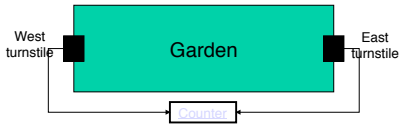
Structure of concurrent programs

Concurrent programs are intrinsically more complex than single-threaded programs:

- when more than one activity can occur at a time, program execution is necessarily nondeterministic;
- code may execute in surprising orders—any order that is not explicitly ruled out is allowed
- a field set to one value in one line of code in a process may have a different value *before the next line of code is executed in that process;*
- ***writing concurrent programs requires new programming techniques***

Example: the Ornamental Garden (Burns & Davis 1993)

A large ornamental garden is open to members of the public who can enter through either of two turnstiles



- the owner of the garden hires a student to write a ***concurrent program*** to count how many people are in the garden at any one time
- the program has two processes, each of which monitors a turnstile and increments a shared counter whenever someone enters via that processes' turnstile

Example continued

• In a simple-minded approach, when a person passes a turnstile, the turnstile reads the counter value, increments it and writes it back.

• This may result in a race condition (east and west turnstile racing to write to the counter) and some people will not be counted.

```
West          counter          East
enter           0            enter
person1                      person2
read counter = 0          read counter = 0
write counter = 1
                1        write counter = 1
                1
```


Module aims

This course introduces the basic principles of concurrent programming and their use in designing programs

Aims

- to convey a basic understanding of the concepts, problems, and techniques of concurrent programming
- to show how these can be used to write simple concurrent programs in Java
- to develop new problem solving skills

Module objectives

- judge for what applications and in what circumstances concurrent programs are appropriate;
- design concurrent algorithms using a variety of low-level primitive concurrency mechanisms;
- analyse the behaviour of simple concurrent algorithms with respect to safety, deadlock, starvation and liveness;
- apply well-known techniques for implementing common producer-consumer and readers-and-writers applications, and other common concurrency problems; and
- design concurrent algorithms using Java primitives and library functions for threads, semaphores, mutual exclusion and condition variables.

Scope of the module

- will focus on concurrency from the point of view of the *application programmer*;
- we will focus on problems where concurrency is implicit in the problem requirements;
- we will only consider imperative concurrent programs;
- we will focus on programs in which process execution is asynchronous, i.e., each process executes at its own rate; and
- we won't concern ourselves with whether concurrent programs are executed in parallel on multiple processors or whether concurrency is simulated by multiprogramming.

Outline syllabus

The course focuses on four main themes:

- introduction to concurrency;
- design of simple concurrent algorithms in Java;
- correctness of concurrent algorithms; and
- design patterns for common concurrency problems.

Assessment

Assessment is by coursework and examination:

- coursework worth 25%, due on the 20th of March; and
- a two hour examination, worth 75%.

There are also several unassessed exercises.

Reading list

- Andrews (2000), *Foundations of Multithreaded, Parallel and Distributed Programming*, Addison Wesley.
- Lea (2000), *Concurrent Programming in Java: Design Principles and Patterns*, (2nd Edition), Addison Wesley.
- Goetz (2006), Java concurrency in practice, Addison Wesley.
- Ben-Ari (1982), *Principles of Concurrent Programming*, Prentice Hall.
- Andrews (1991), *Concurrent Programming: Principles & Practice*, Addison Wesley.
- Burns & Davis (1993), *Concurrent Programming*, Addison Wesley.
- Magee & Kramer (1999), *Concurrency: State Models and Java Programs*, John Wiley.

The next lecture

Processes and Threads

Suggested reading for this lecture:

- Andrews (2000), chapter 1, sections 1.1–1.2;
- Ben-Ari (1982), chapter 1.

Suggested reading for the next lecture:

- Bishop (2000), chapter 13;
- Lea (2000), chapter 1.

Sun Java Tutorial, Threads

`java.sun.com/docs/books/tutorial/essential/threads`