Concurrent Algorithms
Lecture 1 - Introduction

Eugene Kenny

eugkenny.lit@gmail.com

Limerick Institute of Technology

September 15th 2015

Learning Outcomes

- Evalaute methods for synchronising concurrent processes
- Synthesise concurrent programming techniques and assess effects of concurrency in specific domains
- Design algorithms that execute on multi-core and/or multi-processor systems

Indicative Syllabus

- Introduction to Concurrent Computing
- Processes and Synchronisation
- Locks and Barriers
- Semaphores
- Monitors

Assessment

Assessment is by:

- Final Examination: 50%
- Continous Assessment: 50%

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.

Why Concurrency?

- It is often useful to be able to do several things at once:
  - when latency (responsiveness) is an issue, e.g., server design, cancel buttons on dialogs, etc.;
  - when you want to parallelise your program, e.g., when you want to distribute your code across multiple processors;
  - when your program consists of a number of distributed parts, e.g., client-server designs.

But I only have a *Single* Processor

- Concurrent designs can still be effective even if you only have a single processor/core:
  - 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 elses program)
  - even if your code is CPU bound, it can still be more convenient to let the scheduler (e.g. JVM) work out how to interleave the different parts of your program than to do it yourself
  - it is also more portable, if you do get another processor/more cores

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

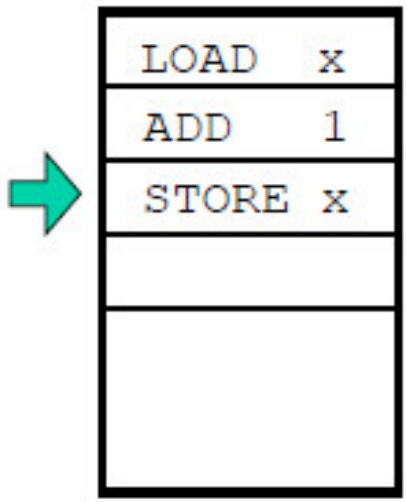
More Examples of Concurrency

- **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.

Sequential Programs

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

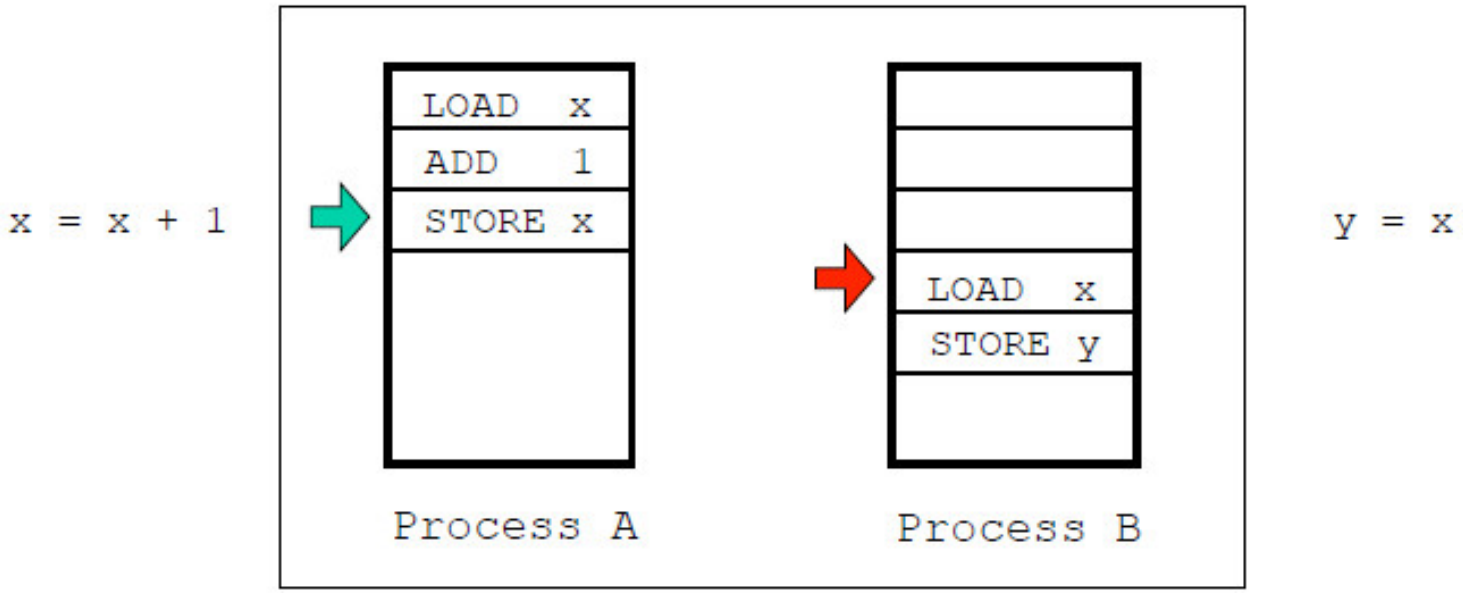
```
x = x + 1
```



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.

Aspects of concurrency

We can distinguish between:

- whether the concurrency is *required* (by the specification) or *optional* (a design choice made by the programmer);
- the *granularity* of the concurrent program, application or system; and
- how the concurrency is *implemented*.

Concurrency in Specification vs Implementation

Concurrency is useful both when we want a program to do several things at once, and as an implementation strategy:

- in real-time systems concurrency is often implicit in the *specification* of the problem, e.g., cases where we cant allow a single thread of control to block on I/O;
- in parallel programming, e.g., weather forecasting, SETI@home, etc., there may be no concurrency in the problem requirements - however a concurrent *implementation* may run faster or allow a more natural problem decomposition.

Granularity of Concurrency

The processes in a concurrent program (or more generally, concurrent application or concurrent system) can be at different levels of granularity:

- threads within a single program, e.g., Java threads within a Java program running on a JVM (lightweight processes);
- programs running on a single processor or computer (heavyweight or OS processes) - this is not usually a concern for applications programmers;
- programs running on different computers connected by a network.

Implementations of Concurrency

We can distinguish 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; and
- **distributed processing:** the execution of concurrent processes by running them on separate processors - processes communicate by message passing.

Java Implementations of Concurrency

Java supports both shared memory and distributed processing implementations of concurrency:

- **shared memory:** multiple user threads in a single Java Virtual Machine - threads communicate by reading and writing shared memory locations; and
- **distributed processing:** via the `java.net` and `java.rmi` packages - threads in different JVMs communicate by message passing or remote procedure call

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 dont 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 dont 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 cant 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.

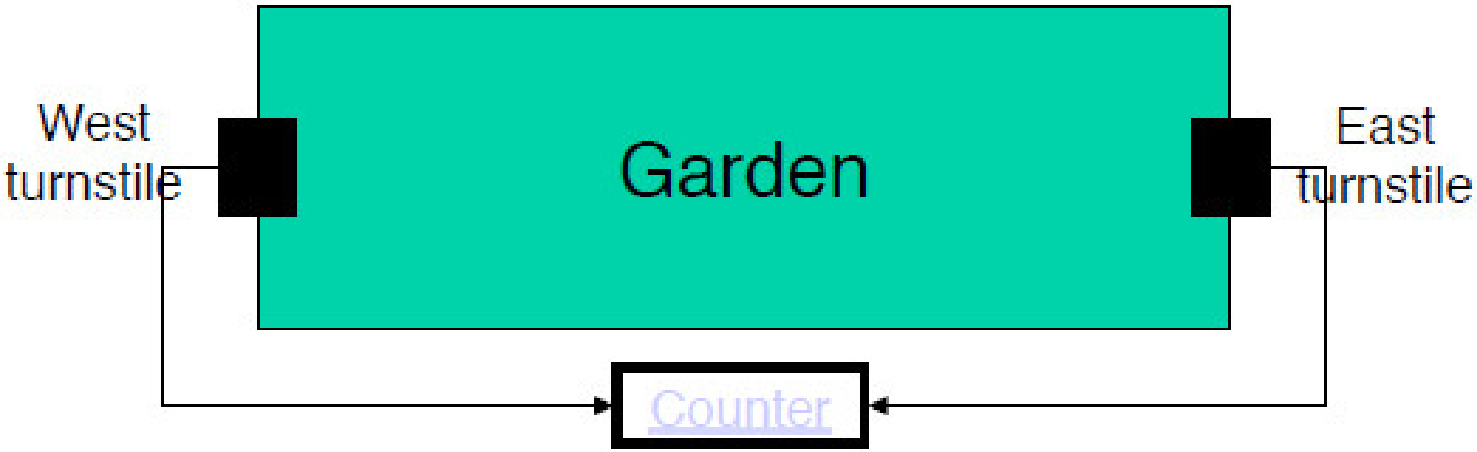
Structure of Concurrent Programs

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 ordersany 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 Gardens Problem

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

Why Study Concurrent Programming?

Advantages of concurrency

- concurrent programs are often more flexible or responsive than single-threaded programs
- a concurrent design is more natural for many applications
- a concurrent design may make better use of system resources

Potential problems of concurrency

- concurrent programs are often more complex than single-threaded programs
- a concurrent design may entail a performance overhead (though this is often over-stated)
- concurrency introduces new kinds of bugs - testing is much less useful in identifying defects