6SENG006W Concurrent Programming

Week 1

Introduction to Concurrency & Modelling

Introduction to Concurrency & Modelling

The aim of this lecture is to introduce some of the basic issues & concepts in concurrency & how we can model them.

The topics we will cover are:

- Explain why *concurrency* is important.
- Role play & discuss two simple concurrent systems:
  - Updating a *Student's Bank Account*.
  - The classic "*Dining Philosophers*" problem.
- Outline some of the problems that can arise in concurrency, e.g. *race conditions*.
- Describe how these problems e.g. *race conditions*, can be solved using concurrent programming features.
- Explain what it means for a concurrent program to be *correct*, e.g. by satisfying *Safety* & *Liveness* properties.
- Example of modelling a simple system using *FSP*.

PART I

Why is Concurrency Important?

Why Concurrency is Important in Computer Science

The main reasons for studying & investigating *concurrency* are:

- The *world is naturally concurrent*, in particular almost all large computer systems involve some form of concurrent activity, e.g. the internet, operating systems (Windows, Unix/Linux, Android), etc, etc.
- Concurrency allows more *efficient use of resources*, *improves speed of execution* & *increases computing power*.
- Concurrency is often the only way to solve problems which involve a *temporal* or *nondeterministic* or *non-sequential* aspect.
- Concurrency, in particular *concurrent programming* & *concurrency theory* are important topics in Computer Science & Software Engineering, e.g. 2 *Turing Awards* – Hoare (CSP) & Milner (CCS).
- Concurrent programming distinguishes *operating systems* & *real-time systems* from other (boring?) types of software systems.
- *You* will probably have to design & implement concurrent or real-time systems for various types of computer systems.

Why Concurrency is Difficult

Concurrency is **Very Difficult** because:

> *the advantages of using concurrency are almost always offset by having to deal with the dramatic increase in complexity of concurrent programs & systems.*

In other words – *"There's no such thing as a free lunch."*

For example, for the two types of programs:

Sequential program: just one thing happening at a time.

Concurrent program: lots of things happening at once.

From the real world, consider *juggling*:

> *1 ball easy, 2 balls okay, ..., **5 or 6 balls v. hard.***

Managing this *added complexity* & applying the appropriate *principles* & *techniques* necessary for the construction of well-behaved (Java) concurrent programs is what this module is about.

Understanding Concurrent Programming is a *Matter of Life or Death!*

Consider the following real case, taken from the set book, see also https://en.wikipedia.org/wiki/Therac-25

Between June 1985 & January 1987, *Therac-25* a radiation therapy machine caused *six known accidents with resultant deaths & serious injuries*.

Errors in the concurrent program controlling the Therac-25 played an important part in these six accidents.

Interactions between different concurrent activities in the control program resulted in *occasional* erroneous control outputs, i.e. doses of x-rays..

The *sporadic nature* of the errors (*"race conditions"*) caused by faults in the concurrent program **significantly delayed** the discovery of the problem.

The *designers* of the Therac-25 software were *largely unaware of the principles & "best" practices* of concurrent programming.

In general, errors in concurrent applications & systems may not always be directly life-threatening but they adversely affect our quality of life & may have severe financial implications.

Conclusion

An understanding of:

- the *issues & problems that can arise in a concurrent program*,
- the *principles & techniques of concurrent programming* &
- an appreciation of *how it is practised*.

is **extremely important in its own right**.

But is an **essential part** of the *“skill set”* of a:

- Computer Scientist,
- Professional Software Engineer &
- anyone who considers themselves a serious & high ability programmer.

PART II

New Problems that arise in the Concurrent World

New Problems that arise in the World of Concurrency

New problems arise when you move from a *sequential* world to a *concurrent* world.

That is, from

"one thing happening at a time"

to

"many things happening at the same time".

We shall now role play the following two example concurrent systems:

1. Updating a *Student's Bank Account*.

2. The *Dining Philosophers* Problem.

These simple concurrent systems will allow us to investigate many of the *issues*, *problems* & *solutions* that arise in concurrent programming.

What to Consider

As we role play these two examples you should consider the following questions:

- *What problems are occurring?*
- *How can these problems be dealt with?*
- *What are the elements/parts of the solution?*
- *What has the solution achieved?*

Student's Bank Account

Consider the following scenario of various updates to a new student Rik's *bank account* on his first day at University.

- *Initially* Rik has a balance of £100.
- The student loan company *deposits* Rik's loan of £1,000 for the term into his account.
- Rik *treats himself* to a new iPad for £400.
- Rik is his Gran's favourite grandchild, so she also *deposits* £500 for the term.
- The Bank Manager makes *no charge* if an account is in **credit**, but to protect his *"bonus"* he will charge £50 a day, if an account goes **overdrawn**.

Question: what will the *balance* be in Rik's Bank Account at the end of the day?

The Character's Actions

Each of the four characters:

- Rik,
- Loan Company,
- Gran
- Bank Manager

performs the same sequence of actions:

1. *Read the bank balance* from the bank.
2. *Calculate the new balance* that results from their transaction.
3. *Write the bank balance* back to the bank.

What can happen?

If every thing goes *okay* –

$$\begin{aligned} balance &= 100 + 1000 - 400 + 500 \\ &= 1200 \end{aligned}$$

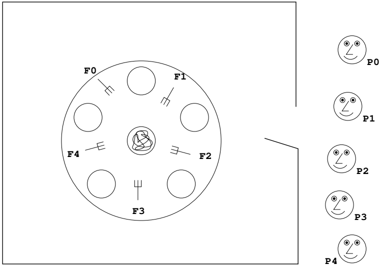
Then Rik's balance is **$\pounds$1200** & he can spend all day playing Minecraft!

But if not then his balance could be **-$\pounds$450** & he'll have to *get a job* to pay off his **overdraft**!

The Dining Philosophers Problem

The *Dining Philosophers* problem is originally due to E.W. Dijkstra (1965) & later developed further by C.A.R. Hoare.

The *Dining Philosophers* problem is used to illustrate various problems that can occur when several (5) processes are competing for limited resources.



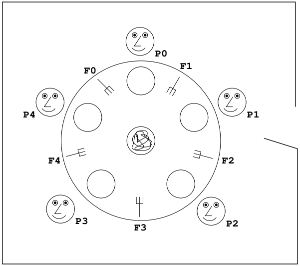


Figure : 1.1 Dining Philosophers Problem

Initially, no philosophers are in the dining room, but they can all sit at the table at once.

The Dining Philosophers Problem Description

Given the layout of the Dining Philosopher's Table (Fig. 1.1) the problem is as follows:

- There are usually *5 philosophers* who spend their time *alternating between thinking & eating*.
- After they have *thought* for a while they *get hungry* & *enter the dining room* & *sit at their allotted place* at a round table.
- In the middle of the table is a *bowl of spaghetti*.
- Between *each pair of philosophers* is *one fork*.
- Before an individual philosopher can take & eat some spaghetti he *must have two forks* – one taken from the *left* & one taken from the *right*.
- The philosophers *must find some way to share forks* such that they *all get to eat*.

A Philosopher's Behaviour

The behaviour of each philosopher is a cycle:

1. *think*,
2. when they get hungry they *enter the dining room*,
3. *sit down* at their *allotted place*,
4. *pickup the fork* on their *right* (their fork),
5. *pickup the fork* on their *left* (their neighbour's fork),
6. *eat* spaghetti,
7. *put down* the fork on their *right*,
8. *put down* the fork on their *left*,
9. *leave* the dining room.

Role Play Dining Philosophers Problem

We'll now do a demonstration that can *deadlock* & one that is *deadlock free*.

You can also watch this YouTube video (after skipping the ads):

"Gary Explains - Dining Philosophers Problem with Solution" [8:09 mins]:
https://www.youtube.com/watch?v=NbwbQQB7xNQ

Better still, try role playing the problem at home with your family or 4 friends, see the Dining Philosophers Problem pack on Blackboard.

- First get into groups of 5.
- Pick up a *Dining Philosophers Pack*.
- Follow the philosopher's cyclic behaviour.
- See what happens & take notes.

What Happened?

Some of the following issues may have arisen during the scenario:

- *Shared objects* & the associated problem of *interference*, if concurrent activities are allowed free access to such objects.
- This leads to the need for *mutually exclusive access to shared objects*.
- So we need to use some form of *coordination*, i.e. *synchronising* their actions.
- Checking the required properties of concurrent programs – *"absence of deadlock"*, where the program **stops & makes no further progress**.
- *Correctness properties* are generally described as either:
  - *"Safety Properties"* – a program **not** reaching a **bad state**, or
  - *"Liveness Properties"* – a program **eventually** reaching a **good state**.

PART III

Concurrency Concepts & Issues

Concurrency Concepts & Issues

We will now look at some of the concurrency concepts & issues that might have arisen during the role playing of these two scenarios.

- *Race Conditions*
- *Synchronisation & Synchronisation Mechanisms*
- *Protection Mechanisms & "Critical Sections"*
- *Interference*
- *Mutual Exclusion*
- *Timing*
- *Correctness of Concurrent Programs*

Race Conditions

When a number of *processes are executed in parallel* the *speed* at which they execute can not be determined.

Therefore, if data must be *transferred between* them this *"nondeterminism"* must be dealt with.

Example

If one process is *waiting for input* from two other processes it can not be determined before hand which will produce it first.

This is an example of a *"race condition"*.

A mechanism must be introduced to *coordinate & control the order* in which processes execute.

These types of problems are usually solved by the use of a *synchronisation mechanism*.

E.g. access to Rik's Bank Account, 2 philosophers going to pick up the same fork.

The "Synchronisation Mechanism"

A *"synchronisation mechanism"* is used to *coordinate* the actions of one or more processes.

A *synchronisation mechanism* must be able to deal with a *dynamic* or *time varying* situation.

E.g. the *Butler* or my *"magic pen"*.

And should have the following properties:

- The ability to *detect actions performed* by the *interacting processes*.
- The ability to *permit the transfer of data* at the *appropriate time*.

These two properties require that communication between dependent processes & their synchronisation are *always linked* & usually are integrated in one mechanism.

Protection Mechanisms & “Critical Sections”

A synchronisation mechanism must be augmented by a *protection mechanism* to stop *interference* between concurrently executing processes.

This means that access to **shared resources** such as variables, line printers, etc,

> ***must ALWAYS be restricted to ONE process at a time.***

This is the purpose of a protection mechanism.

The sequence of statements in a process which require exclusive use of a shared resource is called a *“critical section”* or *“critical region”*.

Example *critical sections*:

- picking up a fork,
- updating Rik’s Bank Account.

Avoiding Interference

Interference between concurrently executing processes is *avoided* by ensuring that:

- **No two processes are executing their critical sections at the same time.**
- Once a process is executing its critical section other processes are **barred entry into theirs** by means of a **"locking"** arrangement.
- When a process leaves its critical section other processes are **permitted to enter theirs** by means of an **"unlocking"** arrangement.

The *access to a critical section* follows this pattern:

waiting → **lock** → *execute critical section* → *unlock* → *continue*

E.g. updating the Bank Account, via *possession of the "magic pen"*.

Mutual Exclusion

Mutual Exclusion is one of the most important problems in concurrent programming because it is an abstraction of many synchronisation problems.

Definition of Mutual Exclusion

> *If activity a of process P* ***must not overlap*** *with activity b of process Q then we say that a & b are required to be mutually exclusive.*

In particular, if *P* & *Q simultaneously attempt* to execute *a* & *b* then we must ensure that *only one of them succeeds*.

The *losing process must be blocked*, i.e. it must not proceed until the winning process completes the execution of its activity.

Mutual Exclusion = “Locks” + “Protocols”

Desirable to make both the *critical section* & the *protocol* as short as possible to maximise concurrency & avoid “problems”.

(See Deadlock.)

Timing

No assumptions are **EVER** made about the *absolute* or *relative* execution speeds of processes or processors.

The reason for this strict rule is that, failure to adopt it leads to serious errors in system design & programming.

Reasons for ignoring timing:

- intuition is not able to cope with time scale involved, e.g. milli-, micro- & nanoseconds.
- time dependent bugs are extremely difficult (impossible) to detect & correct.
- dynamic nature of computer configurations, e.g. faster processors, etc.

There are formal models of concurrency, e.g. *Timed CSP*, that support reasoning about timing information of concurrent systems.

However, these formal systems are usually very complicated to use.

Correctness of Concurrent Programs

Concurrent programs are *notorious* for the hidden & very sophisticated bugs they contain.

Question:

"What does it mean for a concurrent program to be correct?"

The correctness properties of *sequential* programs are known as:

Partial correctness: if the program terminates then it will produce the correct result.

Total correctness: the program will terminate & it will produce the correct result.

The notion of correctness for a *concurrent* program is different to that for a sequential program.

Correctness of Concurrent Programs

As with sequential programs we shall say that:

> *A concurrent program is correct if it satisfies its specification.*

But the specification of a concurrent program is very different to that of a sequential program.

We specify the correct behaviour of a concurrent program by using two categories of what are called *"desirable properties"*.

These two categories are known as:

Safety Properties — *"nothing bad ever happens"*

Liveness Properties — *"something good eventually happens"*

Examples of “Safety” Properties

We now give some examples of classes of desirable *safety* properties:

- ensure *“mutual exclusion”* — process has exclusive access to the critical region.
  E.g. Rik’s bank account.

- avoid *“deadlock”* — processes do not get stuck.
  E.g. each philosopher can eventually pick up their forks & eat.

Examples of “Liveness” Properties

We now give some examples of classes of desirable *liveness* properties:

- avoid *“livelock”* – processes make progress, i.e. do not get stuck in a cycle of pointless activities.
  E.g. philosophers endlessly picking up/putting down a fork.
- ensure *“accessibility”* – processes can get access to the resources they need. E.g. forks, Rik’s bank account.
- ensure *“fairness”* – all processes are given a chance to make progress.
  E.g. all philosophers have an opportunity to pick up forks.
- avoid *“individual starvation”* – no process is continuously blocked, by other processes.
  E.g. philosophers don’t all gang up on Donald J. Trump.

PART IV

A Very Brief Introduction to Finite State Processes (FSP)

Introduction to FSP

The topics we will briefly cover are:

- *FSP* view of a *process* and a *concurrent program*.
- Small example of an *FSP* process that *models* a drinks vending machine – `DRINKS`.
- *Modelling* a concurrent program by a *Finite State Machine (FSM)*.
- Illustrate how to use the modelling tool – *LTS Analyzer* (`ltsa`) to:
  - *analyse* the `DRINKS` process,
  - *animate* the `DRINKS` process (i.e. execute it),
  - draw a *graphical representation* of the `DRINKS`.

Note: these topics will be the subject of the next three lectures.

FSP's Abstract Process Model

FSP has a very *abstract high-level* view of a process & how individual processes can be combined in parallel.

An Individual Process

We will use an *abstract model* of a *process*, where:

- a process has a *state*;
- its *state* is modified by *indivisible* or *atomic actions*;
- actions cause *transitions* from *one state to another*.

A Concurrent Program

A *concurrent program* is than modelled by a *collection of* these abstract processes executing in *parallel*.

FSP Example: A DRINKS Vending Machine

The following example *FSP process* describes a drinks dispensing machine that can dispense:

- hot `coffee` if the **`red`** button is pressed &
- iced `tea` if the **`blue`** button is pressed.

DRINKS

```
DRINKS = (   red  -> coffee -> DRINKS
           | blue -> tea    -> DRINKS
         ) .
```

Where:

- `DRINKS` is the *name* of the *process*;
- **`red`**, `coffee`, **`blue`** & `tea` are *actions*;
- "`->`" *action prefix* operator;
- "`|`" *choice* operator.

DRINKS Graphical Representation

The graphical representation of the DRINKS finite state machine is given in Fig. 1.2.

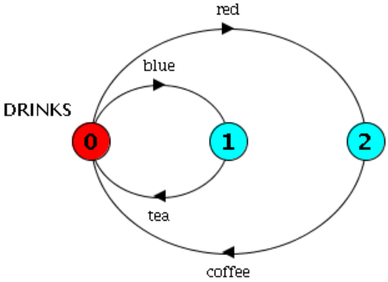


Figure : 1.2 DRINKS state machine.

Where:

- the *states* are labelled 0, 1, 2;
- 0 is the *initial state*;
- it has 4 *transitions* labelled with the *actions*: **red**, coffee, **blue** & tea.

Accessing & Opening an FSP Process in the LTSA Tool

To be able to use & load an FSP process into the LTA tool you need to do the following:

1. LTSA is implemented as a Java Archive `jar` file.
2. Download the `ltsa.jar` file to your system.
3. To start up LTSA tool:

   ```
   java -jar ltsa.jar &
   ```

4. Load the FSP `DRINKS` process into the tool.
   Either by typing it into the tool directly using the *Edit* tab; or if it exists in a text file `Drinks.lts` then *Open* it from the *File* menu.

Animate & Analyse an FSP Processes using the LTSA Tool

To be able to analyse & animate a process or collection of processes using the LTSA tool you need to do the following:

1. To animate the program it must be *parsed* & *compiled*.
2. So from the *Build* menu first select *Parse* then *Compile*.
   Feedback – error messages, etc, are displayed in the *Output* tab.
3. To draw `DRINKS` process' LTS, use *Draw* tab & select it.
4. To animate/execute the `DRINKS` process, either click "*A*" on toolbar or use menu *Check* > *Run* > *DEFAULT*.
5. The *Animator* pop-up window lets the user control the actions offered by the LTS model to its environment.
   - Executable actions are *ticked* (✓).
   - Displays the *"trace"*, i.e. the sequence of actions performed.