SWEN90004
Modelling Complex Software Systems

Concurrency

Nic Geard

Lecture Con.01

Semester 1, 2019

Concurrent programs

A sequential program has a single thread of control, that is, a single instruction pointer suffices to manage its execution.

A concurrent program allows multiple threads of control. This enables multiple calculations to occur at the same time, as well as simultaneous interaction with external events.

Each thread, or process, in a concurrent program often either shares data or communicates with one or more other threads in that program.

Parallel processing

Parallel processing

Historically, interest in concurrency stems from the need to program parallel hardware.

The late 1950 saw the introduction of special processors, called data channels, to control devices such as tape drives, card readers and printers, with the ability to run simultaneously with the central general-purpose processor.

It was soon realised that synchronization of processors and managing access to shared resources, however, is a hard problem.

Operating systems were devised to handle this complexity.

Parallelism vs concurrency

Concurrency is best thought of as a design principle—structuring programs to reflect potential parallelism.

A program written in a concurrent programming language may be executed with or without actual parallelism; on a single processor typically by time-sharing.

Conversely, a program written in a sequential programming language may be executed with parallelism; for example, via vectorization.

Why study concurrency?

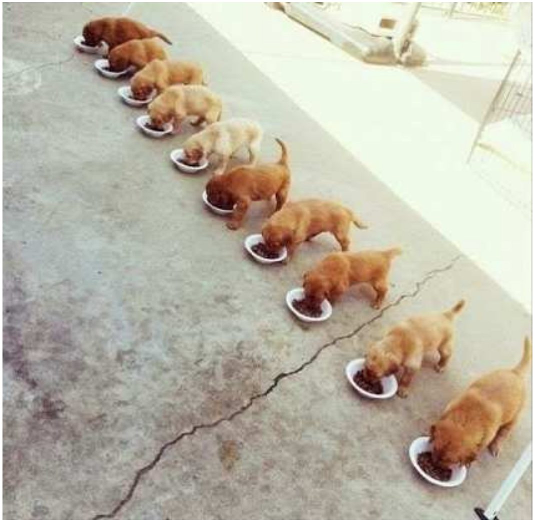
- A natural model: Modelling software using a concurrent structure may better reflect the reality in which the software runs. Consider a user interface using the keyboard, mouse and multiple windows.
- Necessity: In some domains, it is unavoidable. To control an autonomous robot with multiple moving parts, multiple threads are required to ensure all parts can react to sensor stimulation.
- Performance: Executing a program concurrently can give us increased performance, especially if multiple processors are available.

Concurrent programming is hard

The reason why concurrent programming is hard is that processes need to interact:

- Communication: Processes generally need to communicate with each other, either by accessing shared data, or by message passing.
- Synchronization: Processes may need to synchronize certain events, such as “P must not reach point p until after Q has reached point q.”

Concurrency: theory



Concurrency: practice

Non-determinism

In fact designing, implementing, testing and debugging concurrent software systems are all hard.

A major source of frustration is the fact that the execution of a concurrent program is non-deterministic.

This explains the huge interest in tools such as model checkers, that can help to formally establish important properties of concurrent programs, such as safety and liveness.

(more about these later...)

Concurrent programming language paradigms

Concurrent languages roughly fall in two categories:

The shared-memory type, utilising the concept of monitors.

These have a long history from Concurrent Pascal to Java and C#.

The message-passing type, based on Hoare's idea of Communication Sequential Processes (CSP).

Examples include Occam, Erlang and Go.

In this part of the subject . . .

We will introduce the main concepts in concurrency, and explore the concurrent features of a shared-memory programming language, Java, in tutorials and assignment 1a.

Then we will introduce a concurrency modelling language, FSP, and show how it can be used for design and analysis of concurrent systems, in particular verification of safety and liveness properties. We will explore FSP over the course of several tutorials and assignment 1b.

We'll then come back to look at how modelling in FSP can help us to write more correct concurrent code in Java, plus look at an alternative message-passing approach to programming concurrency using Go.

Speed dependence

Unlike sequential programs, a concurrent program may be speed-dependent—its behaviour may depend on the relative speeds of its components' execution.

Small, random fluctuations in processor or input-output speed are sources of observed non-determinism.

When the absolute speed of a system and its components matters—typically in embedded systems—we talk about real-time systems.

Arbitrary interleaving

The usual model of concurrent behaviour is that, at the level of atomic events, no two events happen exactly at the same time.

Assume process P performs the atomic events a, b, in that order.

Assume process Q performs x, y, z.

There are 10 possible ways to interleave the two sequences, while maintaining order.

The "arbitrary interleaving" model says that these 10 sequences are exactly the possible outcome of running P and Q concurrently.

Arbitrary interleaving: Motivation

The advantage of the “arbitrary interleaving” model is that it allows us to ignore real time.

This abstraction makes programs more amenable to formal analysis.

If we can develop verification tools that establish program properties under this model then those properties are invariant under changes to hardware.

Quiz: What to expect here?

Assume we run these two processes concurrently.

They have access to a shared variable, *n*, which is initialized to 0.

```
process P:
    int i;
    for i := 1 to 10 do
        n := n+1;
```

```
process Q:
    int i;
    for i := 1 to 10 do
        n := n+1;
```

When both processes have completed, what is the value of *n*?

Quiz: What to expect here?

Assume we run these two processes concurrently.

They have access to a shared variable, *n*, which is initialized to 0.

```
process P:
    int i;
    for i := 1 to 10 do
        n := n+1;
```

```
process Q:
    int i;
    for i := 1 to 10 do
        n := n+1;
```

When both processes have completed, what is the value of *n*?

The answer depends on what the atomic actions are.

Atomicity

WYSINWYX: What you see is not what you execute!

In most programming languages an assignment such as `n := n+1` is not atomic; a compiler will break it up into more basic instructions:

```
process P:
    int p
    do 10 times:
        load n into p
        increment p
        store p back in n
```

```
process Q:
    int q
    do 10 times:
        load n into q
        increment q
        store q back in n
```

So, when both processes have completed, what is the value of n?

Interference and atomic instructions

A possible start of an interleaving is this:

```
load n into p   // P makes its register 0
load n into q   // Q makes its register 0
increment p     // P makes its register 1
increment q     // Q makes its register 1
store p  in n   // P makes n 1
store q  in n   // Q makes n 1
```

The processes go wrong because each falsely assumes to have exclusive access to `n` in its read-change-write cycle.

This is known as interference or a race condition. It justifies our interest in mutual exclusion, a topic we will study in the next lecture.

Interference

Running the two processes ten times in a row gave these results:

```
n is: 13
n is: 12
n is: 15
n is: 11
n is: 14
n is: 13
n is: 14
n is: 16
n is: 14
n is: 15
```

What are the possible values that `n` can take?

Interference

Interleaving causes problems because there are many different possible interleavings that can occur, and each execution may yield a different interleaving. However:

> *A concurrent program must be correct for all possible interleavings.*

From the example above, we get quite different behaviour.

To make matters worse, on different machines, we may get drastically difference in behaviour.

Loop from before, unrolled

```
process P:
   int p
     1  load n into p
     2  increment p
     3  store p back in n
     4  load n into p
     5  increment p
     6  store p back in n
     7  load n into p
     8  increment p
     9  store p back in n
    10  load n into p
    11  increment p
    12  store p back in n
    13  load n into p
    14  increment p
    15  store p back in n
```

```
process Q:
   int q
     1  load n into q
     2  increment q
     3  store q back in n
     4  load n into q
     5  increment q
     6  store q back in n
     7  load n into q
     8  increment q
     9  store q back in n
    10  load n into q
    11  increment q
    12  store q back in n
    13  load n into q
    14  increment q
    15  store q back in n
```

Safety and liveness properties

We distinguish between safety and liveness properties of concurrent systems.

Roughly, safety means that "nothing bad will ever happen" and liveness means that "something good eventually happens".

Interference (or rather its absence) is an archetypal safety property.

Deadlock (or rather its absence) is an archetypal liveness property.

Summary

Concurrency is potential parallelism.

Concurrency is an abstraction that makes it easier to reason about the dynamic behaviour of a system.

Formal approaches to understanding concurrent systems are required, because their behaviour is often non-deterministic.

In a concurrent program, atomic operations can be interleaved arbitrarily.

For a concurrent program to be correct, it must be correct for all possible interleavings.

References / Recommended reading

M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, Prentice Hall, 2nd edition, 2006.

E. Dijkstra, *Cooperating Sequential Processes*, manuscript, 1965. `www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF`.