# SWEN90004
# Modelling Complex Software Systems

## Wrapping up

Nic Geard

Review Lecture

Semester 1, 2019

# Topics: Concurrency

Concurrency is a design principle for structuring programs to reflect potential parallelism.

Compared to a sequential system, a concurrent system is characterised by multiple threads of control (sometimes referred to as processes).

These processes need to:

- communicate: either via shared memory (Java) or message passing (FSP, Go)
- synchronise: ie, one thread can't proceed until another is in a certain state

Development and debugging are difficult due to nondeterminism.

# Topics: Modelling concurrency

When reasoning about concurrent systems, we abstract away absolute time and consider relative timing: the order in which atomic events occur.

Atomic events may be arbitrarily interleaved (occur in any order).

We typically want to ensure that concurrent programs behave correctly for any arbitrary interleaving.

# Topics: Interference

We illustrated the problems that can arise when concurrent processes interfere with each other; eg if two processes each try to execute the following atomic events:

- read shared value `x`
- update value `x = x+1`
- write shared value `x`

# Topics: Mutual exclusion

We studied a simple mutual exclusion scenario in which two threads each alternated between a non-critical and a critical section of code (in which the shared value was accessed).

Solutions needed to avoid:

- deadlock/livelock: when one or more threads want to enter their CS, one must succeed
- starvation: a thread wanting to enter its CS must eventually succeed

# Topics: Concurrency in Java

Java handles concurrency using threads, created by extending `Thread` or implementing `Runnable`.

A Java thread that is alive can be in one of several states:

- runnable (ready to be executed)
- running (currently being executed)
- non-runnable (eg, sleeping, waiting on another thread, etc)

# Topics: Concurrency in Java

We indicate to the Java VM that a thread should be executed by calling `start()`. This causes the VM to execute the thread's `run()` method in its own concurrent thread. The thread will stop when `run()` finishes.

Calling `t.join()` suspends the calling thread until `t` has finished running.

Threads can be interrupted while non-runnable, in which case they will return to being runnable and throw an `InterruptedException`.

# Topics: Monitors

A monitor is a construct that handles synchronization.

It provides threads with:

- mutual exclusion in access to shared (private) data
- the ability to wait for a certain condition to become true
- (and to be notified when that condition becomes true)

While a thread has (exclusive) access to a monitor, it is said to hold that monitor's lock.

# Topics: Synchronisation in Java

Any object in Java can act as a monitor; all objects have:

- an intrinsic lock, which is obtained by a thread when it calls a `synchronized` method, and released when that method returns
- a means of handling which threads are currently waiting to obtain that lock

We can mark a method as a critical section by declaring it with the `synchronized` keyword.

# Topics: Synchronisation in Java

Methods relevant to synchronisation in Java:

- `wait()`: Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.
- `notify()`: Wake up a single (arbitrary) thread waiting on this object's lock.
- `notifyAll()`: Wake up all objects that are waiting on this object's lock.

# Topics: Synchronisation in Java

`wait()` should be wrapped within a `while` loop that checks on the condition it is waiting for:

```
class Account {
    public synchronized void withdraw(int amount) {
        while (balance < amount) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        ...
    }
    ...
}
```

# Topics: Semaphores

Manages access to a shared resource; consists of:

- some number ($v$) of currently available permits
- a wait set ($W$) of processes waiting for permission

`S.wait()`

```
if S.v > 0
    S.v--
else
    S.W = S.W U {p}
    p.state = blocked
```

`S.signal()`

```
if S.W == { }
    S.v++
else
    choose q from S.W
    S.W = S.W \ {q}
    q.state = runnable
```

# Topcis: Concurrency in FSP

FSP is an algebraic language for modelling concurrent systems.

An FSP process is defined in terms of sequences of actions.

FSP processes are eqiuvalent to LTS diagrams.

Benefits of formal approaches such as FSP include:

- unamiguous communication about system structure and dynamics
- automated checking of safety and liveness properties

# Topics: Checking safety and liveness in FSP

Two types of property that are of interest in concurrent systems:

- Safety properties: nothing "bad" (eg, deadlock or interference) ever happens during execution
- Liveness properties: something "good" eventually happens during execution (eg, all processes trying to access their critical section eventually do so)

If we model a concurrent process using FSP, we can automatically check that a defined property holds; that is, is true for every possible trace/execution of the model.

# Topics: Message passing

The message passing paradigm for concurrency removes the concept of shared memory (and associated issues).

Rather, concurrent processes communicate with one another by sending and receiving messages.

Message passing can by synchronous or asynchronous.

Synchronous communication behaves similarly to a shared action in FSP: both the sender and receiver must occur at the same time.

Asynchronous communication removes the need for a receiver to be ready to receive before a sender is able to send.

# Topics: Concurrency in Go

Go uses a message passing approach to concurrency (although it does also allow shared memory).

Concurrency entities in Go are:

- go-routines: are very lightweight processes/threads
- channels: named entities that allow data to be sent and received between go-routines