# Contents

# SYST Part 1

Adam Hawley

April 13, 2019

## 1 Lecture 3: Witness Languages Pt.1

### 1.1 Atomicity

A statement is atomic if it cannot be interfered with:

- Either executed completely or not at all

### 1.2 Race Condition

A race condition is where one process "races" ahead. Processes that should be in sequence, should wait for others to complete, but do not. This results in the output not being as expected. This needs to be considered in concurrent programming design.

### 1.3 Interleaving

## 2 Lecture 4: Witness Languages Pt.2

## 3 Lecture 5: Coordination 1

### 3.1 Interacting Processes

Tasks need to share data. They do this by coordinating, here are some examples of tasks which require coordination:

- One at a time through a critical section
- A starts X after B finishes Y
- Replicated tasks need to combine/compare results
- Work needs to be allocated to a manager

There are two approaches to interacting processes: **shared variables** and **message passing**.

#### 3.1.1 Mutual Exlusion — Mutex

**Critical sections** are code that must not be executed by more than one task at a time. Unfortunately implementations of mutual exclusion are often complex and error prone. They also do not easily generalise to $n$ tasks nor do they easilly generalise to more complex problems.

#### 3.1.2 Levels of Support

**Simple primitive**:

- Semaphores — simple processes for guaranteeing mutual exclusion.
- Mutexes (normally provided by the runtime environment/OS so not discussed in detail).

**Control structures**:

- Monitors which are normally provided by the language.

**Semaphores** A semaphore is a non-negative integer together with two primitives: `wait` and `signal`. On creation, a semaphore is initialised to 1 (in the simplest case).

- `signal` — Atomically incrememnts a semaphore.
- `wait` — If the semaphore has a value greater than zero, decrements the value by 1. If the semaphore is equal to 0 then the executing task **blocks**

Blocking is when a task is not runnable. Tasks are unblocked when its semaphore becomes $> 0$. If multiple tasks are blocked on a semaphore, `signal(sem)` will unblock one task chosen non-deterministically.

# 4 Lecture 6: Coordination Pt.2

## 4.1 Shared Variable Coordination

#### 4.1.1 Monitors

A monitor is a control structure that provides **mutual exclusion**. It can also be considered to be "An extension of the conditional critical sections".

#### 4.1.2 Pascal-FC Implementation of Monitors

See lecture for live coding.

Pascal-FC uses a low-level condition variable mechanism with three operators:

**delay** immediately blocks caller task, and releases the monitor

**resume** makes at most one blocked task runnable again

**empty** return true if no blocked tasks exist

### 4.1.3 Ada Implementation of Monitors

In Ada, monitors are called **protected types**. There are no condition variables like there are in Pascal-FC. Instead, synchronisation comes from the use of barriers (guards).

Specifications of protected types can contain:

- Procedures executed under **mutual exclusion**
- Pure functions that are **read only** can execute concurrently with other functions.
- Entries specify **barriers** (or guards).
- Private data of any type.

**Barriers** Barriers are boolean expressions that must evaluate to `true` for the entry to execute. If `false`, the task is blocked. Barriers are re-evaluated on task entry and exit from the **protected object** (**PO**). At most one task can proceed through the barrier.

**Attributes** Attributes in Ada give information about the behaviour of the program, e.g:

```
entry await when await 'count = 4 is ...
```

This releases one task (non-deterministically) when there are 4 waiting, otherwise blocks.

# 5 Lecture 7: Coordination 3

## 5.1 Recapping Java

In Java, there are two approaches to coordination:

- Extend Thread and override run()
- Declare a class that implements Runnable

### 5.1.1 Shared Variable Coordination

In Java, every object has a **lock** which enforces mutual exclusion. The lock can be used in two different ways:

- The `synchronized` method modifier – Produces a monitor
- The `synchronized` statement – Defines a critical region

Conditional Synchronisation uses methods from the top-level `Object` class:

`wait()` Always blocks when executed, but releases the lock.

`notify()` Wakes up another thread waiting on the currently held lock:

- At most one thread is woken
- Thread selection is non-deterministic
- Woken thread **waits** to obtain the lock

`notifyAll()` Wakes up all waiting threads (which access under normal mutual exclusion)

**LIVE CODE SECTION MISSED AS NOT ON RECORDING**

`Join()` is a method called by a main thread to *join* its subthread and wait for it to complete to ensure consistency throughout both threads.

### 5.2 Readers/Writers Problem

Different implementation strategies exist:

- We choose to give priority to waiting writers
- All (new) readers are blocked if any writer is available.

## 6 Lecture 8: Coordination Recap

### 6.1 Shared Variables

#### 6.1.1 Communication

Using variables to share information between multiple processes.

#### 6.1.2 Synchronisation

- Semaphores
- Condition Variables
- Monitors
- Barriers

## 7 Lecture 9: Message Passing 1

### 7.1 Message-Based Coordination

Shared-variable coordination is based on *normal* variable access by multiple processes. Message-passing coordination requires new primitives. Abstractly, these are seen as:

- `send(message)`
- `receive(message)`

A message can never be received (read) before it has been sent (written). This means that `receive(message)` is potentially **blocking**.

## 7.2 Evaluating Message-Based Coordination

There are a number of ways to describe different approaches:

- Models for send and receive
- How tasks are identified
- The relationship between tasks
- Types of messaages that can be sent

## 7.3 Possible Models for Sending

### 7.3.1 Aynchronous

Sender does not wait, `send()` returns *immediately* but sent messages must be buffered.

### 7.3.2 Synchronous

Sender blocks until receiver can receive, vice-versa. Message can pass directly from sender to receiver, this means no buffer is necessary. Sometimes this approach is called *rendezvous*.

### 7.3.3 Remote Invocation

Sender blocks until receiver is ready (Synchronous). This time, when the message is received, the receiver will flag an acknowledgement or reply from the reviver. This is also known as an *extended rendezvous*.

## 7.4 Task Naming

How do senders and receivers refer to each other?

- Direct
- Indirect
- Symmetric
- Asymmetric

#### 7.4.1 Direct

Tasks have names. This is used to identify end points, e.g send(message) to Joe.

#### 7.4.2 Indirect

An intermediate is used (e.g. a *mailbox* or a *channel*). These names can be statically named.

#### 7.4.3 Symmetric

Sender specifies receiver and vice versa. For example:

- `send(message) to task`
- `receive(message) from task`

#### 7.4.4 Asymmetric

Send to named receiver, receive from any sender:

- `send(message) to task`
- `receive(message)`

### 7.5 Task Relationships

What is the relationship between senders and receivers?

- One-to-one (can be synchronous or buffered)
- One-to-many
  - A **multi-cast** communication
  - Sometimes inaccurately called a *broadcast* which is one-to-everybody
  - Requires indirect naming
- Many-to-many (multiple multi-casts)

### 7.6 Message Types

These can be limited to a set of predefined types or unlimited (all types/classes can be communicated). Low-level languages are more likely to limit message type.

**SEE LECTURE FOR OCCAM**

# 8 Lecture 10: Message Passing 2

## 8.1 Pascal-FC

- Synchronous communication
- Unlimited message types
- Indirect naming via channels
- Guarded select statements
- Has an extended rendezvous mechanism

## 8.2 Ada

- **Remote invocation** communication model
  - Can be used to provide **sychronous communication**
- Unlimited message types
- **Direct symmetric** naming via task names, and an entry defined for that task
- Guarded select statements
- Has extended rendezvous

### 8.2.1 Ada Communication Model

Based on a **client/server** coordination model. A **service** is a **public** `entry` in the server's task specification. An `entry` declaration specifies the name, parameters and result types for the service.

### 8.2.2 Other Facilities

- `'count` gives the number of tasks queued on an entry.
- Entry families declare groups of entries
- Nested accept statements allow multiple task coordination
- A task executing in an `accept` can also execute an `entry` call

### 8.2.3 `select`

The select statement comes in four forms:

```
select_statement ::= selective_accept
                     conditional_entry_call
                     timed_entry_call
                     asynchronous_select
```

`selective_accept` This allows the server to:

- wait (for more than one more rendezvous at a time)
- time-out (if no rendezvous occurs within a specified time)
- terminate (if no client can ever call an entry)

### 8.3 Synchronisation

- Both tasks must *agree* to communicate
- **Ready** task **waits** for the other task (blocked)
- When both tasks are ready, client's arguments are passed to the server.
- Server executed code inside the `accept` statement
- Results returned to client at completion of `accept`
- Tasks continue concurrently

I can't tell if there was actually more content or not...

## 9 Lecture 11: Message Passing 3

### 9.1 Erlang

- It is an **eager** functional language.
- Asynchronous message passing communication model
  - **Non-Blocking** send
  - Send never fails (even to non-existent ids)
  - Inter-process **buffers** are **unbounded** - conceptually
  - Sending order is mainitained at receiver
  - Receives from different senders, are read non-deterministically
- **Unlimited** message types
- **Direct asymmetric** naming via **process id** (pid)

#### 9.1.1 Send !

Values of any type can be sent. It has Occam-like syntax (e.g `Pid4 ! {pos, 42}`).

### 9.1.2 `receive`

Each process has an unbounded queue for received messages. The arrival order is maintained for each sender but the buffer is interleaved between senders.

Removing messages from the mailbox is using pattern matching from the oldest entry and it is blocking if no match is available.

Receive so has ways of implementing a select from Ada or ALT from Occam and a guarded select:

```
% Ada Select or Occam ALT
receive
    pattern1 -> expression1;
    pattern2 -> expression2;
    pattern3 -> expression3;
end

% Guarded Select
receive
    pattern1 when guard1 -> expression1;
    pattern2 when guard2 -> expression2;
    pattern3 when guard3 -> expression3;
```