<table>
<thead>
<tr>
<th><strong>COMP2212 Programming</strong></th>
<th><strong>Message passing concurrency</strong> (Erlang), <strong>Software transactional memory</strong> (Clojure)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language Concepts</strong></td>
<td></td>
</tr>
</tbody>
</table>

Concurrency in Programming Languages

- Last week we discussed shared memory concurrency and concentrated on Java
  - other imperative OO languages, such as C++ and Objective C handle shared memory concurrency using similar constructs
  - shared memory concurrency is **hard** to get right
- in this lecture we take an overview of some other PL approaches to concurrency
  - Erlang - asynchronous message passing
  - Clojure - immutable data, software transactional memory
Message Passing

- forget about critical regions by not allowing shared memory — all communication is explicit: processes send each other messages
- easier: fewer risks of deadlock, races, visibility bus, etc.
- more general: distributed systems do not have shared memory
- increasing hardware support for message passing in many/multi core hardware
Synchrony vs Asynchrony

- Synchronous communication
  - sender blocks until receiver is ready to receive
  - popular e.g. in languages for hardware design
  - synchronous behaviour is difficult (sometimes impossible!) to guarantee in a distributed setting

- Asynchronous communication
  - sender sends and continues execution
  - callbacks
  - popular in distributed applications
Example - Producer/Consumer

```c
#define N 100

void producer() {
    int item;
    message m;

    while (1) {
        item = produce_item();
        receive(consumer, &m); // receive "empty slot" message
        build_message(&m, item);
        send(consumer, &m);
    }
}

void consumer() {
    int item, i;
    message m;

    for (i=0; i<N; i++) receive(producer, &m); // send N "empty slot" messages
    for (i=0; i<N; i++) {
        receive(producer, &m);
        item = extract_item(&m);
        send(producer, &m); // send "empty slot" message for message consumed
        consume_item(item);
    }
}
```
Erlang

- **Ericsson Language** - designed for distributed, fault-tolerant applications
- functional core, call-by-value, garbage collection
- for concurrency uses Asynchronous Message Passing
  - more accurately, the **actor model**
- increasingly widespread in distributed application (e.g. powers Facebook chat)
**Actor model**

- each process is considered to be an **actor**
- each actor has an address and a message queue (typically FIFO)
- actors can send messages to each other’s addresses where they are enqueued on the message queue until processed
Spawning Threads

http://www.erlang.org/download/getting_started-5.4.pdf

```erlang
-module(tut14).
-export([[start/1, say_something/2]]).

say_something(What, 0) -> done;

say_something(What, Times) ->
    io:format("~p~n", [What]),
    say_something(What, Times - 1).

start() ->
    spawn(tut14, say_something, [hello, 3]),
    spawn(tut14, say_something, [goodbye, 3]).
```

Spawns two threads, the first writes “hello” three times, the second “goodbye” three times. The interleaving is nondeterministic and depends on the scheduler.
Message passing

```erlang
-module(tut15).
-export([start/0, ping/2, pong/1]).

ping(0, Pong_PID) ->
Pong_PID ! finished,
io:format("ping finished\n", []).

ping(N, Pong_PID) ->
Pong_PID ! {ping, self()},
receive
  pong ->
    io:format("Ping received pong\n", [])
end,
ping(N - 1, Pong_PID).

pong() ->
  receive
    finished -> io:format("Pong finished\n", []);
    {ping, Ping_PID} ->
      io:format("Pong received ping\n", []),
Pong_PID ! pong,
pong()
end.

start() ->
Pong_PID = spawn(tut15, pong, []),
spawn(tut15, ping, [3, Pong_PID]).```
Names

Name is created before spawning ping

register allows you to register PIDs
name lookup is done at runtime
receive Construct

receive
  pattern1 ->
    actions1;
  pattern2 ->
    actions2;
...
  patternN ->
    actionsN
end.

Uses pattern matching that chooses different code depending on message currently processed.
Example

```erlang
-module(tut16).
-export([start/0, ping/1, pong/0]).

ping(0) ->
    pong ! finished,
    io:format("ping finished~n", []).

ping(N) ->
    pong ! {ping, self()},
    receive
    pong ->
        io:format("Ping received pong~n", [])
        end,
    ping(N - 1).

pong() ->
    receive
    finished ->
        io:format("Pong finished~n", []).
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
        end.

start() ->
    register(pong, spawn(tut16, pong, [])),
    spawn(tut16, ping, [3]).
```
Distributed programming with Erlang

- Nodes have names
- Addresses can incorporate nodes as well as registered names
- Simple distributed computation across a network
Software Transactional Memory

- Inspired by **Database Transactions**
  - databases face the same concurrency problems as concurrent programs!
- Optimistic concurrency (assume that nothing will go wrong)
  - during a **transaction** all changes are made
  - there is a **commit**
    - if successful (ie. there has been no interference) the changes are validated
    - otherwise there is a **rollback**
- main idea: all or nothing
  - if transaction successful, all changes have to be done together atomically
  - if transaction unsuccessful, no changes are made
- simple for the programmer: no explicit locking
Example

```java
void deposit(account, amount) {
    lock(account);
    int t = bank.get(account);
    t = t + amount;
    bank.put(account, t);
    unlock(account);
}
```

```java
void deposit(account, amount) {
    atomic {
        int t = bank.get(account);
        t = t + amount;
        bank.put(account, t);
    }
}
```

❖ A higher level **declarative** abstraction - we don’t specify how atomicity is achieved in the code

❖ When are concurrent transitions correct?

❖ **serialisability** - changes, globally, should look as if transactions were executed in series, even if the transactions were performed concurrently
Challenges

- How to detect conflict?
- How to deal with **side-effects** during a transaction
  - How can we rollback a `printf`?
- How to deal with **nested** transactions?
Multiversion concurrency control

- First arose in database management
- Everyone sees a snapshot of a database at any particular moment in time
  - readers do not see concurrent writes until after they have been committed
  - readers do not block writers
  - writers do not block readers
- Old versions of data do not get immediately destroyed, relies on persistent data structures
- How to implement without unrealistic amounts of copying?
  - copy-on-write
  - rely on clever designs for individual data structures
Persistent data structures

- All closure data structures are persistent
- Copying is kept to a minimum
- Example: a persistent hashmap
Clojure overview

- designed by Rick Hickey
- like Scheme, can be considered to be a flavour of Lisp
  - syntax similar to Lisp and Scheme (parentheses!)
- compiles to JVM, java interoperability like Scala
  - why are many languages based on frameworks such as JVM or .Net?
- functional
- dynamically typed
- designed for concurrency
- metadata for all types
The Clojure Philosophy

❖ mutable objects are a problem
  ❖ in a concurrent setting eventually create an intractable network of objects with mutable state
  ❖ coordination, locking
  ❖ difficult to test
❖ if you throw mutability away, things become easier
❖ Clujure concurrency
  ❖ if data cannot be altered, life is simple: there are no races
Refs and STM

- Of course the real world is mutable
  - the clojure approach: references change, the data does not!
- Ref - a datatype that holds references
- Refs can only be changed within a transaction
- All changes are atomic and isolated
  - every change to Refs is made or none
  - no one can see each others changes within a transaction
- code within transactions should not have side-effects
STM and Clojure

- Surround code with `(dosync)`
- Uses Multiversion Concurrency Control (MVCC)
  - a snapshot system, readers may see an “old” state of the world
  - readers never block writers
  - writers never block readers
Clojure Agents

- Refs are for coordinated, synchronous changes to multiple locations
- Agents are attached to a single location for their lifetime and perform their changes asynchronously
  - a fn is sent to an agent
  - action is executed asynchronously by a thread pool
- See Hickey’s Ant Colony demo