University of Glasgow

School of Computing Science

60 YEARS OF COMPUTING AT GLASGOW

Message Passing (1)

Advanced Operating Systems

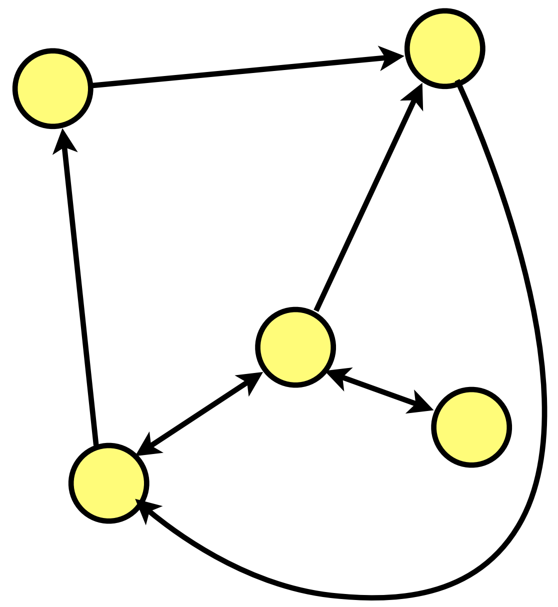
Lecture 11

Lecture Outline

- Message passing systems
  - Approaches and principles
  - Design choices
- Examples
  - Erlang, Scala+Akka
  - Rust
- Avoiding race conditions
  - Immutable data
  - Ownership tracking

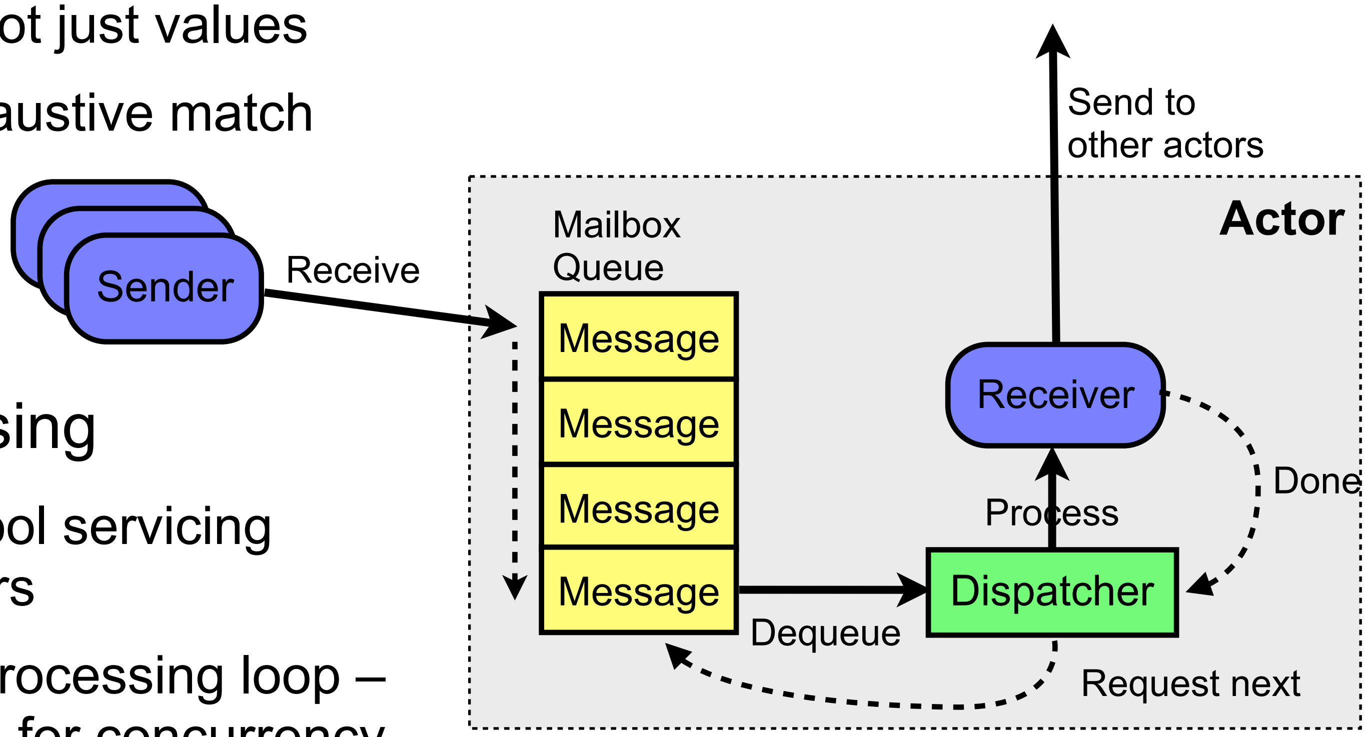
Message Passing Systems

- System is structured as a set of communicating processes, with no shared mutable state
- All communication via exchange of messages
  - Messages are generally required to be immutable – data conceptually copied between processes
  - Some systems use linear types to ensure messages are not referenced after they are sent, allowing mutable data to be safely transferred
- Implementation
  - Implementation within a single system usually built with shared memory and locks, passing a reference to the message – rely on correct locking of message passing implementation
  - Trivial to distribute, by sending the message down a network channel – the runtime needs to know about the network, but the application can be unaware that the system is distributed



Message Handling

- Receivers pattern match against messages
  - Match against message types, not just values
  - Type system can ensure an exhaustive match



- Messages queued for processing
  - Dispatcher manages a thread pool servicing receiver components of the actors
  - Receivers operate in message processing loop – single-threaded, with no concern for concurrency
  - Sent messages enqueued for processing by other actors

Types of Message Passing

- Several different message passing system designs:
  - Synchronous vs asynchronous
  - Statically or dynamically typed
  - Direct or indirect message delivery

- Each has advantages and disadvantages

Interaction Models

- Message passing can involve rendezvous between sender and receiver
  - A synchronous message passing model – sender waits for receiver
- Alternatively, communication may be asynchronous
  - The sender continues immediately after sending a message
  - Message is buffered, for later delivery to the receiver
  - Synchronous rendezvous can be simulated by waiting for a reply

Communication and the Type System

- Statically-typed communication
  - Explicitly define the types of message that can be transferred
  - Compiler checks that receiver can handle all messages it can receive – robustness, since a receiver is guaranteed to understand all messages
- Dynamically-typed communication
  - Communication medium conveys any time of message; receiver uses pattern matching on the received message types to determine if it can respond to the messages
  - Potentially leads to run-time errors if a receiver gets a message that it doesn't understand

Naming of Communications

- Are messages sent between named processes or indirectly via channels?
  - Some systems directly send *messages* to actors (processes), each of which has its own mailbox
  - Others use explicit *channels*, with messages being sent indirectly to a mailbox via a channel

  - Explicit channels require more plumbing, but the extra level of indirection between sender and receiver may be useful for evolving systems
  - Explicit channels are a natural place to define a communications protocol for statically typed messages

Implementations

- Message passing starting to see wide deployment, with two widely used architectures:
  - Dynamically typed with direct delivery
    - The Erlang programming language (https://www.erlang.org/)
    - The Scala programming language (http://www.scala-lang.org) and Akka library (http://akka.io)
    - Dynamically typed – any type of message may be sent to any receiver
    - Messages sent directly to named actors, not via channels
    - Both provide transparent distribution of processes in a networked system
  - Statically typed, with explicit channels
    - The Singularity operating system
    - The Rust programming language (https://www.rust-lang.org/)
    - Use asynchronous statically typed messages passed via explicit channels

Example: Scala+Akka

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props

class HelloActor extends Actor {
  def receive = {
    case "hello" => println("hello back at you")
    case _       => println("huh?")
  }
}

object Main extends App {
  // Initialise actor runtime
  val runtime = ActorSystem("HelloSystem")

  // Create an actor, running concurrently
  val helloActor = runtime.actorOf(Props[HelloActor], name = "helloactor")

  // Send it some messages
  helloActor ! "hello"
  helloActor ! "buenos dias"
}
```

The actor comprises a receive loop that reacts to messages as they're received

Complete program is a collection of actors that exchange messages

Example: Rust

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
  let (tx, rx) = channel();

  thread::spawn(move|| {
    let _ = tx.send(42);
  });

  match rx.recv() {
    Ok(value)  => {
      println!("Got {}", value);
    }
    Err(error) => {
      // An error occurred…
    }
  }
}
```

A unidirectional channel, with transmit and receive sides

Spawn a thread, that sends the number "42" down the channel

Wait to receive data from the channel, process the data or handle the error (e.g., if the channel closed unexpectedly)

Trade-offs

- The two approaches behave quite differently:
    - The Scala+Akka model allows weakly coupled processes to communicate via asynchronous and dynamically typed messages:
        - Expressive, flexible, and extensible actor model
        - Robust framework for error handling via separate processes
        - Relative ease of upgrading running systems via dynamic actor insertion
        - Checking happens at run time, so guarantees of robustness are probabilistic
    - Rust's statically typed message passing provides compile-time checking that a process can respond to messages
        - But, requires more plumbing to connect channels
        - Has more explicit error handling
    - The usual static vs. dynamic typing debate

Avoiding Race Conditions

- Runtime ensures a receiver processes messages sequentially, but it is part of a concurrent system
  - Sending and receiving actors may run concurrently
  - Message data is shared between sender and receiver
- Important to ensure message data is immutable
  - Erlang ensures this in the language → data is immutable
  - Scala+Akka requires programmer discipline → potential race conditions if message data modified after message sent
- Or, at least, never mutated once the message has been sent…

Ownership Transfer

- Alternative to immutability: type system ensures ownership of message data is transferred
- A variable with *linear* type must be used only once; it goes out of scope after use
- Potentially useful when sharing mutable data between threads
  - Implement sharing via a `send` function that takes a linear type for the data to be shared
  - Message data consumed by `send` function and receiver, so can't be used by the sender after message has been sent
  - Data doesn't need to be locked, since it can only be used by one thread at once
- The compiler enforces that linear data is not shared between threads

A variant called *affine types* is used in Rust – data that can be used only once

R. Ennals *et al*, Linear Types for Packet Processing, Proceedings of the European Symposium on Programming, Barcelona, March 2004. http://www.cl.cam.ac.uk/~am21/papers/esop04.pdf

Rust programming language: http://rust-lang.org/

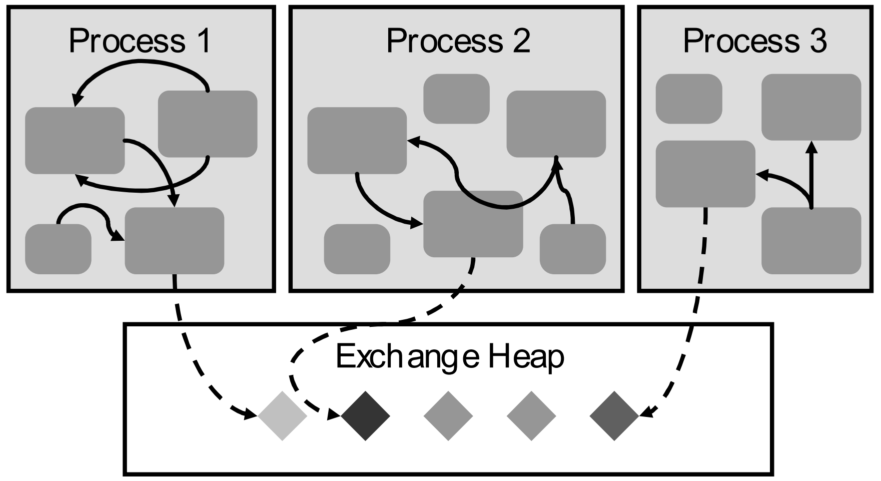
Ownership Transfer: Example

```
use std::sync::mpsc::channel;
use std::thread;

struct State {
    x : i32,
    y : i32
}

fn main() {
  let (tx, rx) = channel();

  thread::spawn(move|| {
    let mut message = Box::new(State {x : 4, y : 2});

    let _ = tx.send(message);

    message.x = 6;
  });

  let result = rx.recv().unwrap();
}

% rustc test.rs
test.rs:15:5: 15:18 error: use of moved value: `message` [E0382]
test.rs:15     message.x = 6;
              ^~~~~~~~~~~~~
```

Race condition avoided – can’t use data after `send()`

Efficiency of Message Passing

- Assuming immutable message or linear types, message passing has an efficient implementation
    - Copy message data in distributed systems
    - Pass pointer to data in shared memory systems
    - Neither case needs to consider shared access to message data



[G. Hunt *et al.*, Sealing OS processes to improve dependability and safety. In Proc. EuroSys 2007, Lisbon, Portugal. DOI 10.1145/1272996.1273032]

- Garbage collected systems often allocate messages from a shared *exchange heap*
    - Collected separately from per-process heaps
    - Expensive to collect, since data in exchange heap owned by multiple threads – need synchronisation
    - Per-process heaps can be collected independently and concurrently – ensures good performance

Summary

- Message passing as an alternative concurrency mechanism
- Increasingly popular
  - Erlang, Scala+Akka (or Java+Akka…)
  - Rust
  - Library-based approaches: ZeroMQ, etc.

- Easy to reason about, simple programming model
  - Provided data is immutable, or ownership is tracked