



University  
of Glasgow

# Concurrency

Advanced Systems Programming (M)

Lecture 6

# Lecture Outline

- Implications of Multicore Systems
  - Memory models
  - Concurrency, threads, and locks
- Alternatives to multithreading
  - Transactions
  - Message passing
    - Immutable data
    - Linear types

# Implications of Multicore

- Memory Models
- Concurrency, threads, and locks

# Memory Models and Multicore Systems

- Hardware trends: multicore with non-uniform memory access
  - Cache coherency expensive → cores communicate by message passing, memory is remote
- When do writes made by one core become visible to other cores?
  - What is the **memory model** for the language?
  - Prohibitively expensive for all threads on all core to have the exact same view of memory (“sequential consistency”)
  - For performance, allow cores inconsistent views of memory, except at synchronisation points; introduce synchronisation primitives with well-defined semantics
- Hardware guarantees vary between processors
  - Differences hidden by language runtime, provided language has a clear memory model

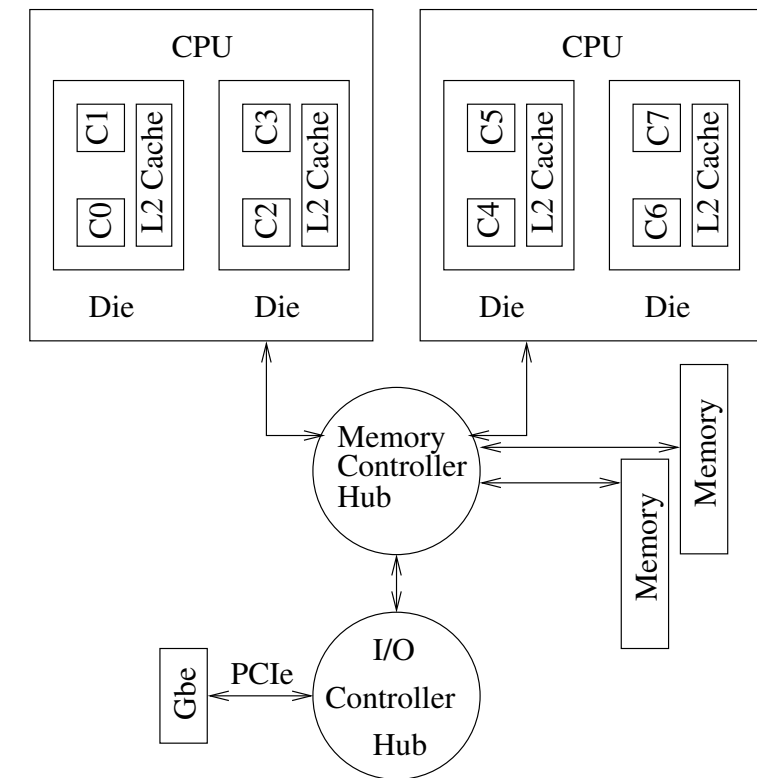


Figure 1. Structure of the Intel system

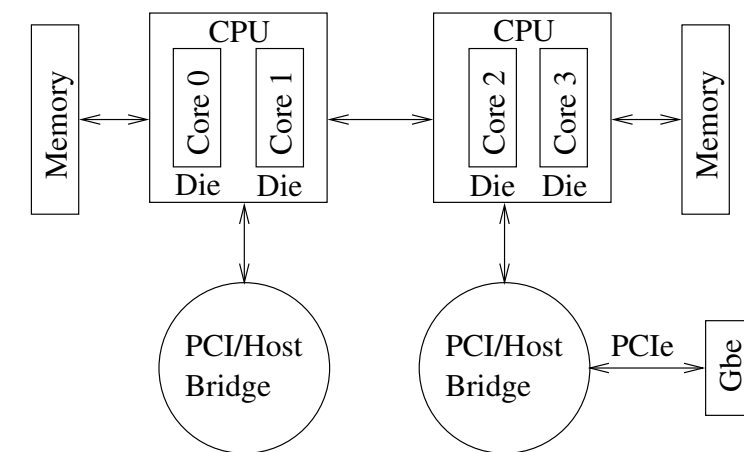


Figure 2. Structure of the AMD system

A. Schüpbach, et al., Embracing diversity in the Barrelfish manycore operating system.  
Proc. Workshop on Managed Many-Core Systems, Boston, MA, USA, June 2008. ACM.

# Memory Models: Java

Java Language Specification, Chapter 17  
<http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf>

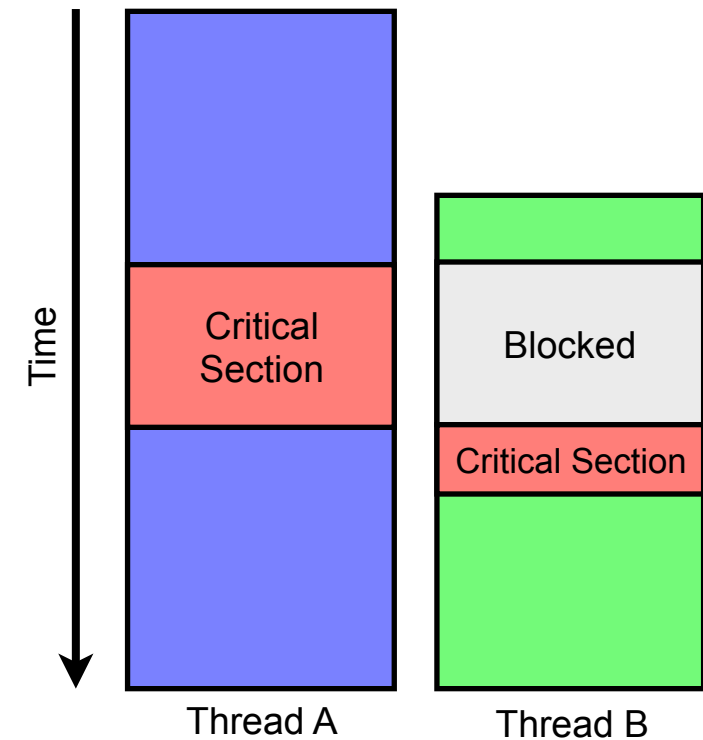
- Java has a formally defined memory model
- Between multiple threads:
  - Changes to a field made by one thread are visible to other threads if:
    - The writing thread has released a synchronisation lock, and that same lock has subsequently been acquired by the reading thread (writes with lock held are atomic to other locked code)
    - If a thread writes to a field declared `volatile`, that write is done atomically, and immediately becomes visible to other threads
    - A newly created thread sees the state of the system as if it had just acquired a synchronisation lock that had just been released by the creating thread
    - When a thread terminates, its writes complete and become visible to other threads
  - Access to fields is atomic
    - i.e., you can never observe a half-way completed write, even if incorrectly synchronised
    - Except for `long` and `double` fields, for which writes are only atomic if field is `volatile`, or if a synchronisation lock is held
- Within a thread: actions are seen in program order

# Memory Models: Others

- Java is unusual in having such a clearly-specified memory model
  - Other languages are less well specified, running the risk that new processor designs can subtly break previously working programs
  - C and C++ have historically had *very* poorly specified memory models – latest versions of standards address this, but not widely implemented
  - Rust does not (yet) have a fully specified memory model
    - Recognised as a limitation – research efforts underway to fix this
    - Complicated by multiplicity of reference types and **unsafe** code

# Concurrency, Threads, and Locks

- Operating systems expose concurrency via *processes* and *threads*
  - Processes are isolated with separate memory areas
  - Threads share access to a common pool of memory
- The processor/language memory models specify how concurrent access to shared memory works
  - e.g., synchronise by explicitly locking critical sections
    - **synchronized** methods and statements in Java
    - **pthread\_mutex\_lock()/pthread\_mutex\_unlock()**
  - Limited guarantees about unlocked concurrent access to shared memory



# Limitations of Lock-based Concurrency

- Major problems with lock-based concurrency:
  - Difficult to define a memory model that enables good performance, while allowing programmers to reason about the code
  - Difficult to ensure correctness when composing code
    - Difficult to enforce correct locking
    - Difficult to guarantee freedom from deadlocks
  - Failures are silent – errors tend to manifest only under heavy load
  - Balancing performance and correctness difficult – easy to over- or under-lock systems




# Composition of Lock-based Code

- Correctness of small-scale code using locks can be ensured by careful coding (at least in theory)
- A more fundamental issue: lock-based code does not compose to larger scale
  - Assume a correctly locked bank account class, with methods to credit and debit money from an account
  - Want to take money from **a1** and move it to **a2**, without exposing an intermediate state where the money is in neither account
  - Can't be done without locking all other access to **a1** and **a2** while the transfer is in progress
  - The individual operations are correct, but the combined operation is not
- This is lack of abstraction a limitation of the lock-based concurrency model, and cannot be fixed by careful coding
- Locking requirements form part of the API of an object

```
a1.debit(v)  
a2.credit(v)
```

Preemption exposes  
intermediate state



# Alternative Concurrency Models

- Concurrency increasingly important
  - Multicore systems now ubiquitous
  - Asynchronous interactions between software and hardware devices
- Threads and synchronisation primitives problematic
- Are there alternatives that avoid these issues?
  - Transactions
  - Message passing

# Managing Concurrency Using Transactions

- Programming model
- Integration into Haskell
- Integration into other languages
- Discussion

# Transactions for Managing Concurrency

- An alternative to locking: use *atomic transactions* to manage concurrency

- A program is a sequence of concurrent atomic actions

- Atomic actions succeed or fail in their entirety, and intermediate states are not visible to other threads

```
atomic {  
    a1.debit(v)  
    a2.credit(v)  
}
```

- The runtime must ensure actions have the usual ACID properties:

- **Atomicity** – all changes to the data are performed, or none are
- **Consistency** – data is in a consistent state when a transaction starts, and when it ends
- **Isolation** – intermediate states of a transaction are invisible to other transactions
- **Durability** – once committed, results of a transaction persist

- **Advantages:**

- Transactions can be composed arbitrarily, without affecting correctness
- Avoid deadlock due to incorrect locking, since there are no locks

# Programming Model

- Simple programming model:
  - Blocks of code can be labelled **atomic** {...}
  - Run concurrently and atomically with respect to every other **atomic** {...} blocks – controls concurrency and ensures consistent data structures
- Implemented via optimistic synchronisation
  - A thread-local transaction log is maintained, records every memory read and write made by the atomic block
  - When an atomic block completes, the log is *validated* to check that it has seen a consistent view of memory
  - If validation succeeds, the transaction *commits* its changes to memory; if not, the transaction is rolled-back and retried from scratch
  - Progress may be slow if *conflicting* transactions cause repeated validation failures, but will eventually occur

# Programming Model – Consequences

- Transactions may be re-run automatically, if their transaction log fails to validate
- Places restrictions on transaction behaviour:
  - Transactions must be referentially transparent – produce the same answer each time they're executed
  - Transactions must do nothing irrevocable:

```
atomic(n, k) {  
    doSomeStuff()  
    if (n > k) then launchMissiles();  
    doMoreStuff();  
}
```

- Might launch the missiles multiple times, if it gets re-run due to validation failure caused by **doMoreStuff()**
- Might accidentally launch the missiles if a concurrent thread modifies **n** or **k** while the transaction is running (this will cause a transaction failure, but too late to stop the launch)
- These restrictions must be enforced, else we trade hard-to-find locking bugs for hard-to-find transaction bugs

# Controlling I/O

- Unrestricted I/O breaks transaction isolation
  - Reading and writing files
  - Sending and receiving data over the networks
  - Taking mouse or keyboard input; changing the display
- Require language control of when I/O is performed
  - Remove global functions to perform I/O from the standard library
  - Add an **I/O context** object, local to **main()**, passed explicitly to functions that need to perform I/O
    - Compare sockets, that behave in this manner, with file I/O that typically does not
    - I/O functions (e.g., **printf()** and friends) then become methods on the I/O context object
    - The I/O context is not passed to transactions, so they cannot perform I/O
    - Example: the IO monad in Haskell

# Controlling Side Effects

- Code that has side effects must be controlled
  - Pure and referentially transparent functions can be executed normally
  - Functions that only perform memory actions can be executed normally, *provided* transaction log tracks the memory actions and validates them before the transaction commits – and can potentially roll them back
    - A *memory action* is an operation that manipulates data on the heap, that could be seen by other threads
    - Tracking memory actions can be done by language runtime (STM; software transactional memory), or via hardware enforced transactional memory behaviour and rollback
- Similar principle as controlling I/O
  - Disallow unrestricted heap access – only see data in transaction context
  - Pass transaction context explicitly to transactions; this has operations to perform transactional memory operations, and rollback if the transaction fails to commit
  - Very similar to the state monad in Haskell



# Monadic STM Implementation (1)

- Monads → way to control side-effects in functional languages
  - A monad **M** **a** describes an action (i.e., a function) that, when executed, produces a result of type **a** performed in the context **M**
    - Along with rules for chaining operations in that context
  - A common use is for controlling I/O operations:
    - The **putChar** function takes a character, operates on the IO context to add the character, and returns nothing
    - The **getChar** operates on the IO context to return a character
    - The main function has an IO context, that wraps and performs other actions
  - The definition of the I/O monad type ensures that a function that is not passed the IO context cannot perform I/O operations
  - One part of a software transactional memory implementation: ensure type of the **atomic {...}** function does not allow it to be passed an IO context, hence preventing I/O

```
putChar :: Char -> IO ()  
getChar :: IO Char
```

# Monadic STM Implementation (2)

- How to track side-effecting memory actions?
  - Define an STM monad to wrap transactions
  - Based on the state monad; manages side-effects via a **TVar** type
    - The **newTVar** function takes a value of type `a`, returns a new **TVar** to hold the value, wrapped in an STM monad
    - **readTVar** takes a **TVar** and returns an STM monad; when performed this returns the value of that **TVar**; **writeTVar** function takes a **TVar** and a value, returns an STM monad that can validate the transaction and commit the value to the **TVar**
- The **atomic** {...} function operates in an STM context and returns an IO context that performs the operations needed to validate and commit the transaction
  - The **newTVar**, **readTVar**, and **writeTVar** functions need an STM action, and so can only run in the context of an atomic block that provides such an action
  - I/O is prohibited within the transaction, since operations in **atomic** {...} don't have access to the I/O context

```
newTVar    :: a -> STM (TVar a)
readTVar   :: TVar a -> STM a
writeTVar  :: TVar a -> a -> STM ()
```

```
atomic :: STM a -> IO a
```

# Integration into Haskell

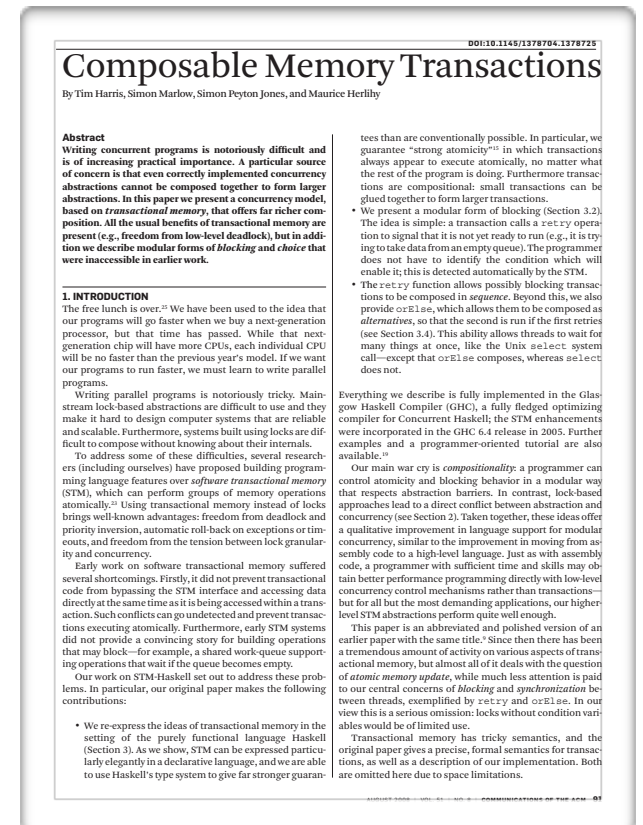
- Transactional memory is a good fit with Haskell
  - Pure functions and monads ensure transaction semantics are preserved
  - I/O and side-effects contained in **STM** monad of an **atomic** {...} block
    - The **TVar** implementation is responsible for tracking side effects
    - The **atomic** {...} block validates, then commits the transaction (by returning an IO monad action to perform the transaction)
  - Untracked I/O or side-effects cannot be performed within an **atomic** {...} block, since there is no way to access the IO monad directly
    - There is no IO monad in scope within the transaction, so code requiring one will not compile
    - A **TVar** requires an **STM** monad, but these are only available in an **atomic** {...} block; can't update a **TVar** outside a transaction, so can't break atomicity guidelines – Haskell doesn't allow unrestricted heap access via pointers, so can't subvert

# Integration into Other Languages

- STM in Haskell is very powerful – but relies on the type system to ensure safe composition and retry
- Integration into mainstream languages is difficult
  - Most languages cannot enforce use of pure functions
  - Most languages cannot limit the use of I/O and side effects
  - Transaction memory can be used without these, but requires programmer discipline to ensure correctness – and has silent failure modes
- Unclear if the transactional approach generalises to other languages

# Further Reading

- Is transactional memory a realistic technique?
- Assumption: shared memory system, doesn't work with distributed and networked systems – is this true?
- Concurrent Haskell:
  - Monadic IO; do notation; IORefs; spawning threads
  - Type system separates state and stateless computation
  - The STM interface
- Do its requirements for a purely functional language, with controlled I/O, restrict it to being a research toy?
- How much benefit can be gained from transactional memory in more traditional languages?



T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy, "Composable Memory Transactions", Communications of the ACM, 51(8), August 2008. DOI:10.1145/1378704.1378725.

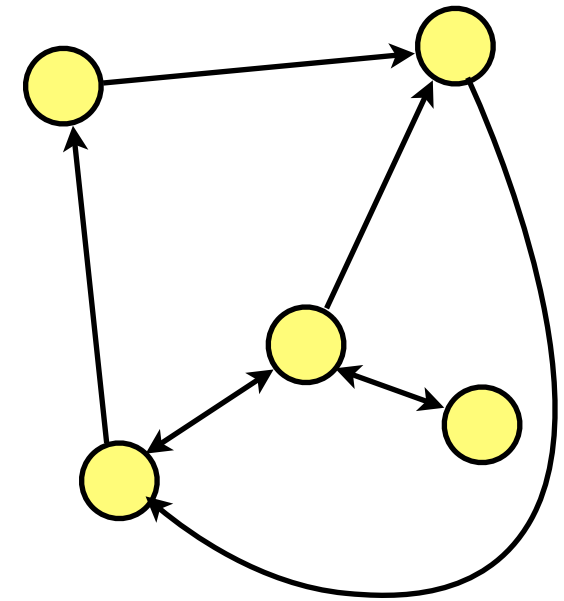
<http://www.cmi.ac.in/~madhavan/courses/pl2009/reading-material/harris-et-al-cacm-2008.pdf>

# Message Passing Systems

- Actors and message passing
- Ensuring safety:
  - Immutable data
  - Control of ownership

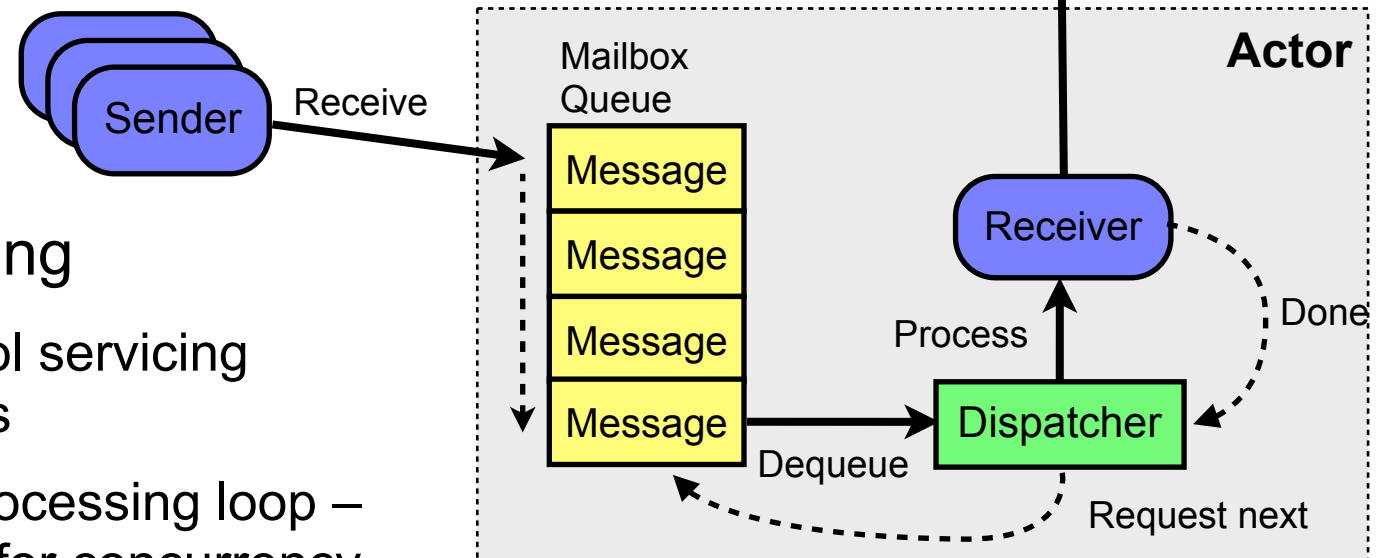
# Message Passing Systems

- System is structured as a set of communicating processes, *actors*, 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

# Types of Message Passing: 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

# Types of Message Passing: Typed Communication

- 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 type 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

# Types of Message Passing: Naming

- 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
    - Erlang programming language (<https://www.erlang.org/>)
    - 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
    - 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...
        }
    }
}
```

# 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...
        }
    }
}
```





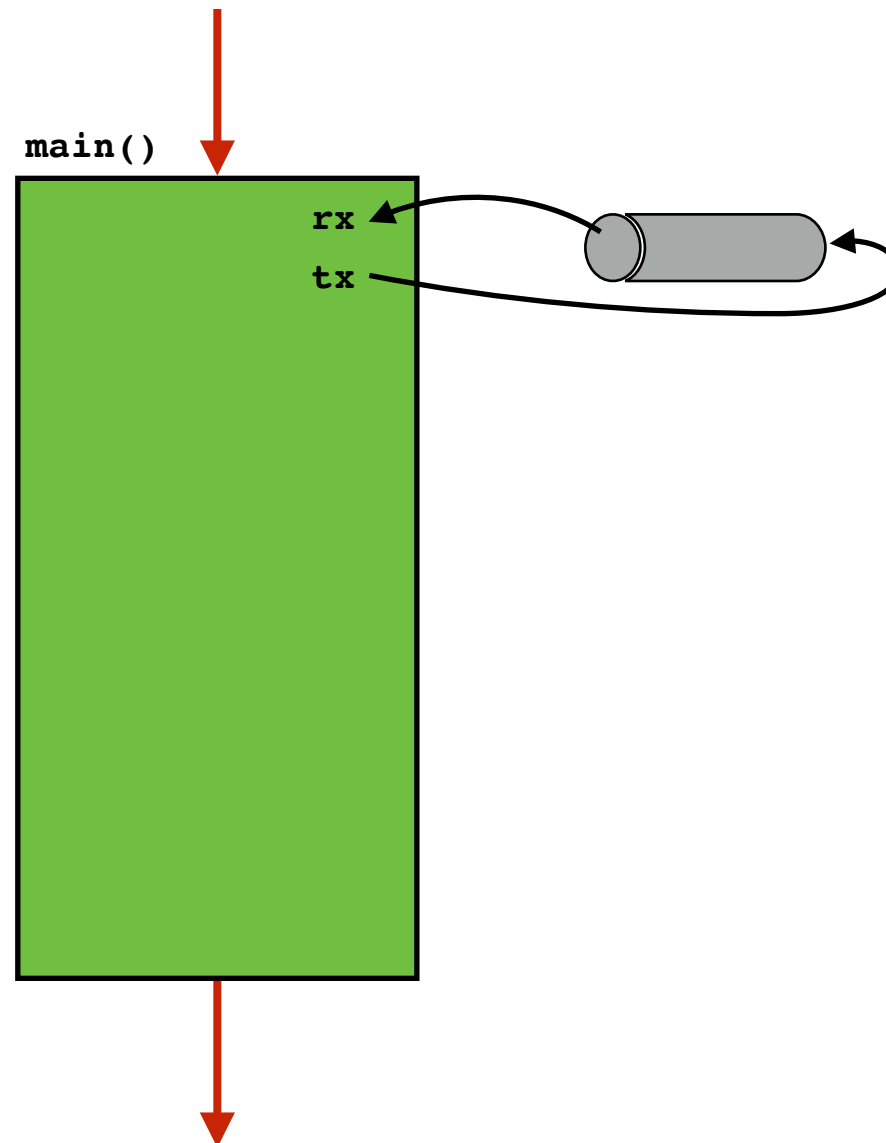
# Example: Rust

```
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...
        }
    }
}
```



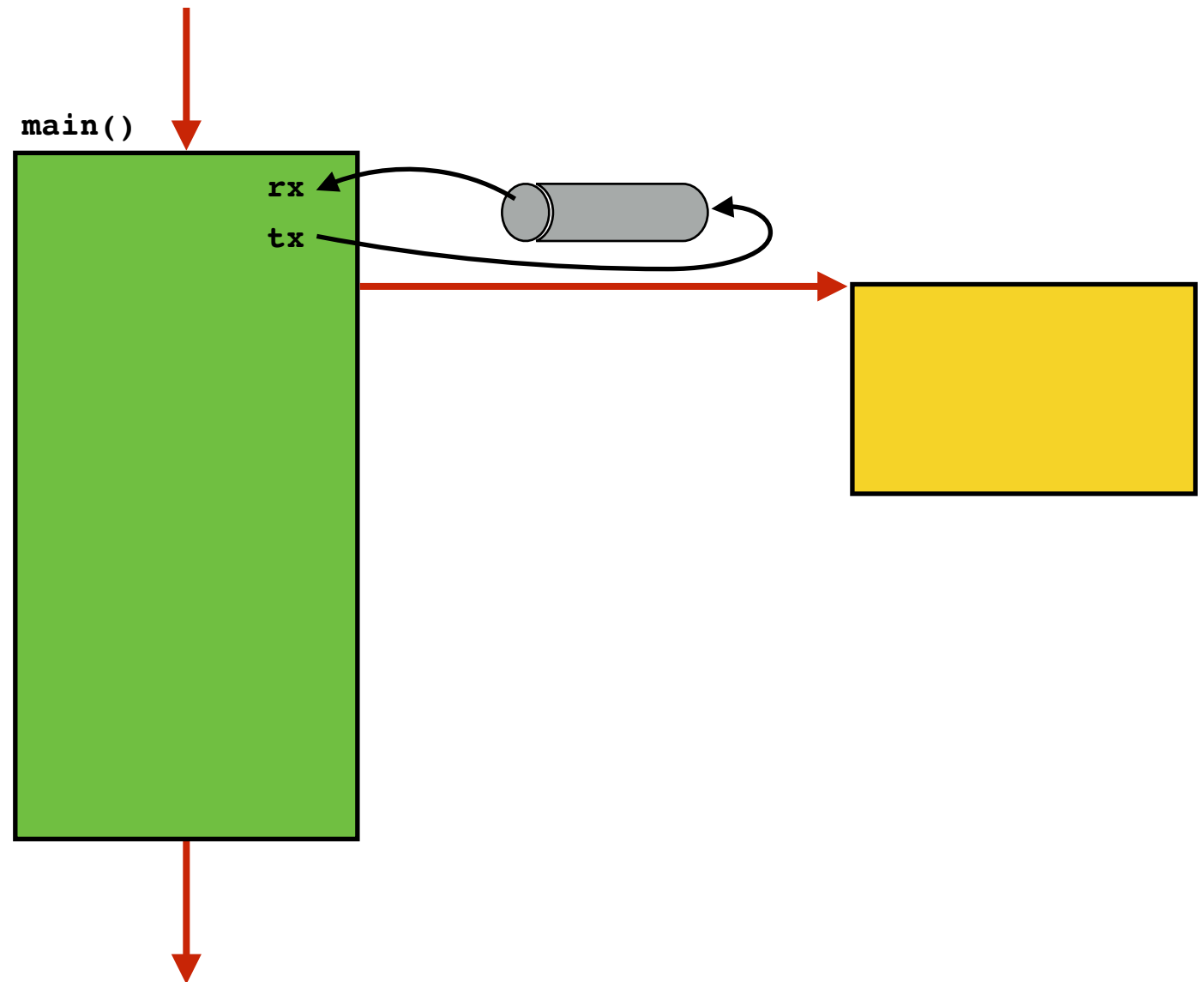
# Example: Rust

```
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...
        }
    }
}
```



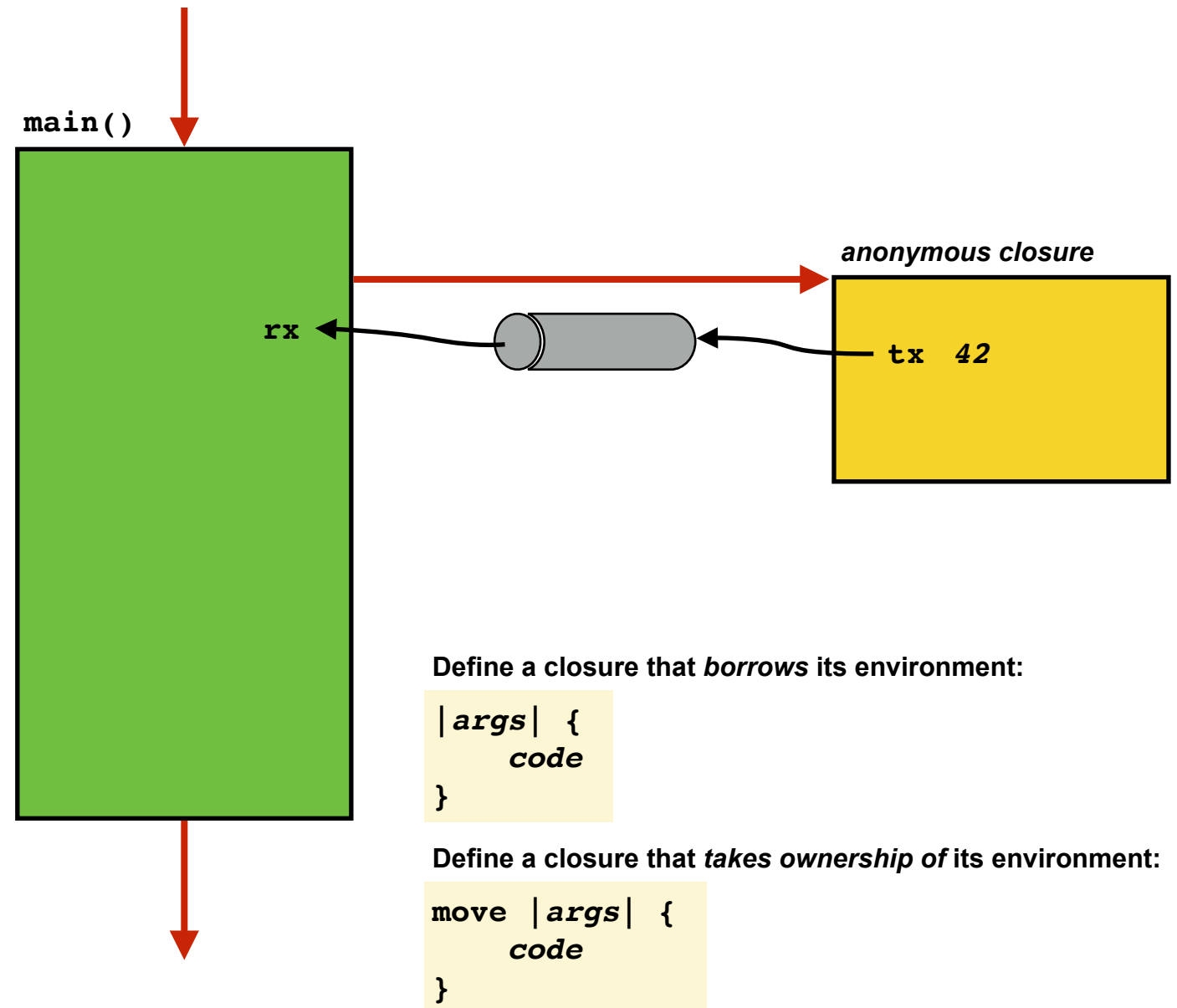
# Example: Rust

```
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...
        }
    }
}
```



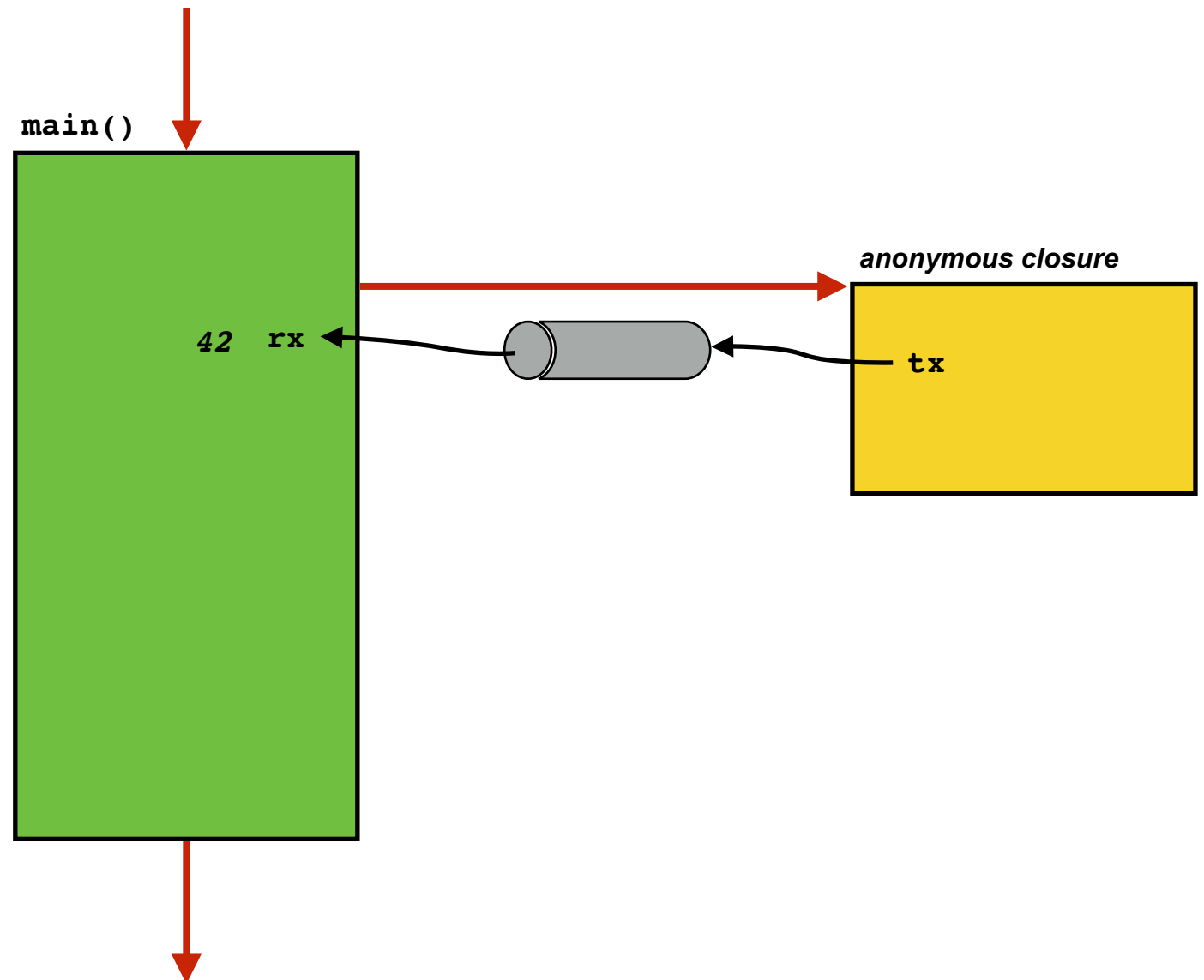
# Example: Rust

```
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...
        }
    }
}
```



# 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 → can only be used by one thread at once
- The compiler enforces that linear data is not shared between threads

# 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;
    });
```

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

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

```
% rustc test.rs
```

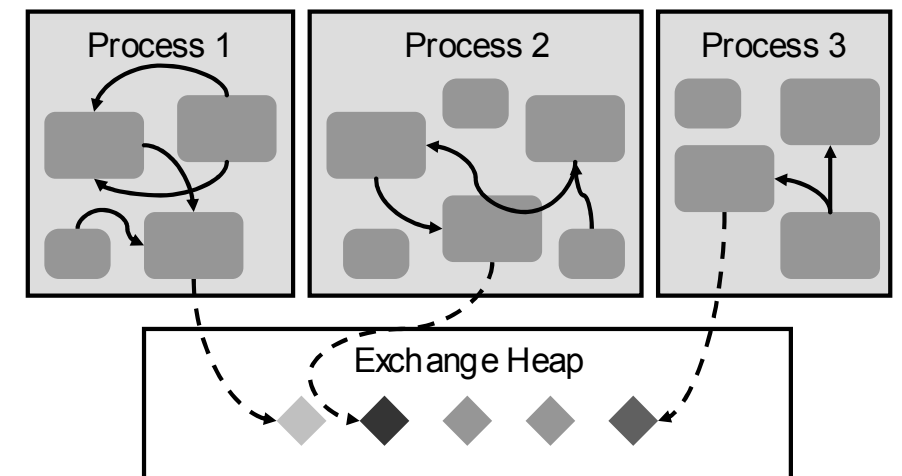
```
test.rs:15:5: 15:18 error: use of moved value: `message` [E0382]
```

```
test.rs:15      message.x = 6;
               ^~~~~~
```



# Efficiency of Message Passing

- Assuming immutable message or linear types, message passing has 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
- 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



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

# Summary

- Message passing as an alternative concurrency mechanism
- Increasingly popular
  - Erlang, Scala+Akka (or Java+Akka...)
  - Rust
  - Go, ZeroMQ, etc. – unchecked message passing
- Easy to reason about, simple programming model
  - Provided data is immutable, or ownership is tracked

# Summary

- Concurrency and memory models
- Transactions
- Message Passing