University of Glasgow

Concurrency

Advanced Systems Programming (M)

Lecture 6

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866,

Lecture Outline

- Implications of Multicore Systems
  - Memory models
  - Concurrency, threads, and locks
- Limitations of multithreading
- Transactions
- Message passing

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Implications of Multicore

- Memory Models
- Concurrency, threads, and locks

BY ND

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

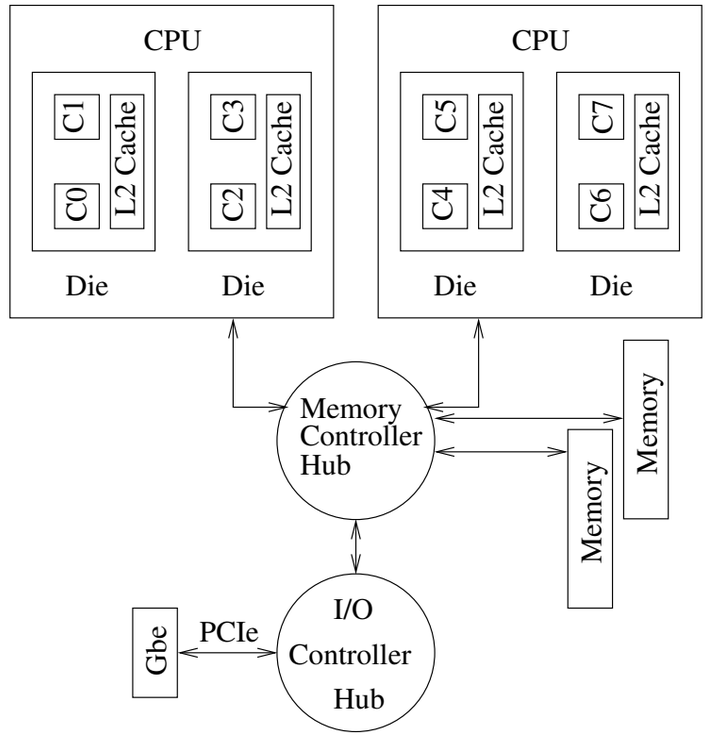


Figure 1. Structure of the Intel system

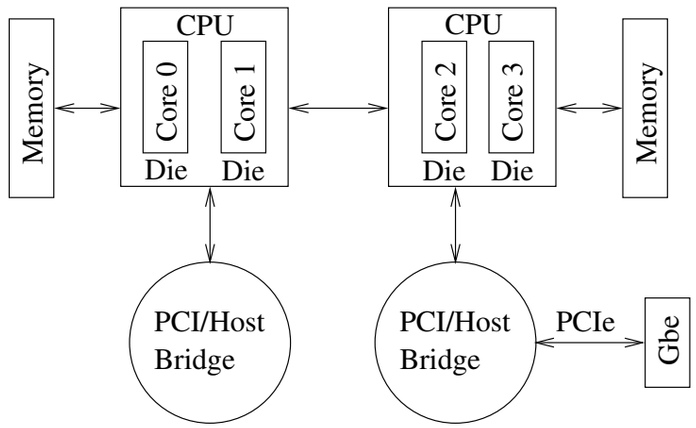


Figure 2. Structure of the AMD system

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.

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Memory Models: Java

- Java has a formally defined memory model

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

  - Changes to a field are seen in program order *within a thread*
  - Changes to a field made by one thread are visible to other threads as follows:
    - If a **`volatile`** field is changed, that change is done atomically and immediately becomes visible to other threads
    - If a non-**`volatile`** field is changed while holding a lock, and that lock is then released by the writing thread and acquired by the reading thread, then the change becomes visible to the reading thread
    - If a new thread is created, it sees the state of the system as if it had just acquired a lock that had just been released by the creating thread
    - If a thread terminates, changes it made become visible to other threads
- Access to all 32-bit fields is atomic
  - i.e., you can never observe a half-way completed write, even if incorrectly synchronised
  - This is not true for for **`long`** and **`double`** fields, which are 64-bits in size, where writes are only atomic if field is **`volatile`** or if a lock is held

CC BY ND

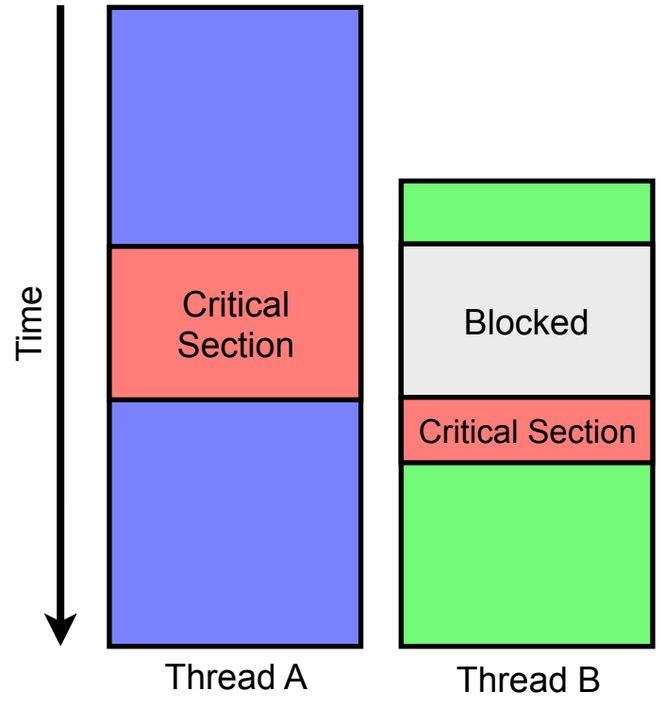
Memory Models: Others

- 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, with memory models heavily influenced by the Java memory model
    - Not yet 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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Concurrency, Threads, and Locks

- Operating system exposes concurrency as **processes** and **threads**
  - Processes are isolated with separate memory areas
  - Threads share access to a common pool of memory
    - Most operating systems started with processes and message passing, and added threads later due to programmer demand
- The memory model specifies how concurrent access to shared memory works
  - Synchronisation by explicit locks around critical sections
    - **`synchronized`** methods and statements in Java
    - **`pthread_mutex_lock()`**/**`pthread_mutex_unlock()`** in C
  - Synchronisation by **`volatile`** fields
  - Limited guarantees about unlocked concurrent access to shared memory



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

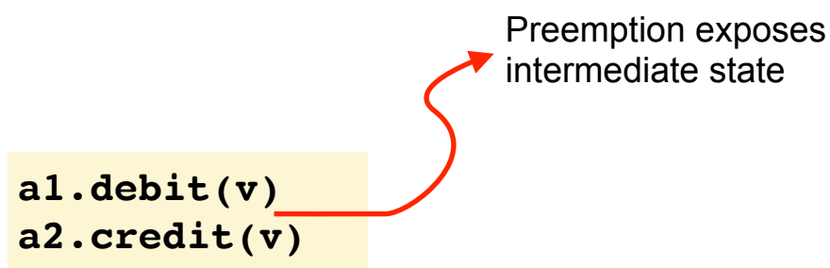
Limitations of Lock-based Concurrency

- 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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Managing Concurrency Using Transactions

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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
  - The runtime must ensure actions have the usual ACID properties:
    - **A**tomicity – all changes to the data are performed, or none are
    - **C**onsistency – data is in a consistent state when a transaction starts, and when it ends
    - **I**solation – intermediate states of a transaction are invisible to other transactions
    - **D**urability – once committed, results of a transaction persist

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

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

CC BY ND

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

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

- 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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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`**

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

- The **`atomic {…}`** function operates in an STM context and returns an IO context that performs the operations needed to validate and commit the transaction

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

  - 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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Integration into Other Languages

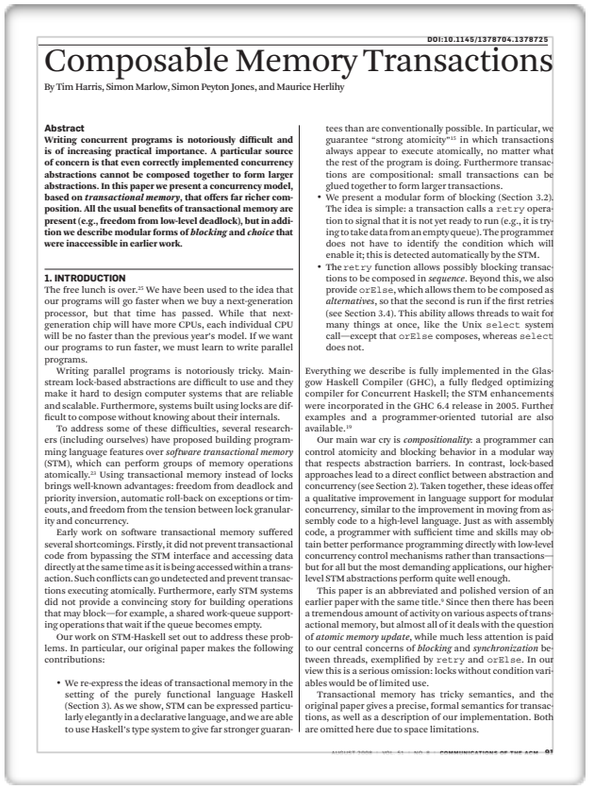
- STM in Haskell is very powerful – but relies on the type system to ensure safe composition and retry
- Integration into mainstream languages is difficult
  - Most languages cannot enforce use of pure functions
  - Most languages cannot limit the use of I/O and side effects
  - Transaction memory can 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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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?



DOI:10.1145/1378704.1378725

Composable Memory Transactions

By Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy

Abstract

Writing concurrent programs is notoriously difficult and is of increasing practical importance. A particular source of concern is that even correctly implemented concurrency abstractions cannot be composed together to form larger abstractions. In this paper we present a concurrency model, based on *transactional memory*, that offers far richer composition. All the usual benefits of transactional memory are present (e.g., freedom from low-level deadlock), but in addition we describe modular forms of *blocking* and *choice* that were inaccessible in earlier work.

1. INTRODUCTION

The free lunch is over.[22] We have been used to the idea that our programs will go faster when we buy a next-generation processor, but that time has passed. While that next-generation chip will have more CPUs, each individual CPU will be no faster than the previous year's model. If we want our programs to run faster, we must learn to write parallel programs.

Writing parallel programs is notoriously tricky. Mainstream lock-based abstractions are difficult to use and they make it hard to design computer systems that are reliable and scalable. Furthermore, systems built using locks are difficult to compose without knowing about their internals.

To address some of these difficulties, several researchers (including ourselves) have proposed building programming language features over *software transactional memory* (STM), which can perform groups of memory operations atomically.[23] Using transactional memory instead of locks brings well-known advantages: freedom from deadlock and priority inversion, automatic roll-back on exceptions or timeouts, and freedom from the tension between lock granularity and concurrency.

Early work on software transactional memory suffered several shortcomings. Firstly, it did not prevent transactional code from bypassing the STM interface and accessing data directly at the same time as it is being accessed within a transaction. Such conflicts can go undetected and prevent transactions executing atomically. Furthermore, early STM systems did not provide a convincing story for building operations that may block—for example, a shared work-queue supporting operations that wait if the queue becomes empty.

Our work on STM-Haskell set out to address these problems. In particular, our original paper makes the following contributions:

- We re-express the ideas of transactional memory in the setting of the purely functional language Haskell (Section 3). As we show, STM can be expressed particularly elegantly in a declarative language, and we are able to use Haskell's type system to give far stronger guarantees than are conventionally possible. In particular, we guarantee "strong atomicity"[21] in which transactions always appear to execute atomically, no matter what the rest of the program is doing. Furthermore transactions are compositional: small transactions can be glued together to form larger transactions.
- We present a modular form of blocking (Section 3.2). The idea is simple: a transaction calls a `retry` operation to signal that it is not yet ready to run (e.g., it is trying to take data from an empty queue). The programmer does not have to identify the condition which will enable it; this is detected automatically by the STM.
- The `retry` function allows possibly blocking transactions to be composed in *sequence*. Beyond this, we also provide `orElse`, which allows them to be composed as *alternatives*, so that the second is run if the first retries (see Section 3.4). This ability allows threads to wait for many things at once, like the Unix `select` system call—except that `orElse` composes, whereas `select` does not.

Everything we describe is fully implemented in the Glasgow Haskell Compiler (GHC), a fully fledged optimizing compiler for Concurrent Haskell; the STM enhancements were incorporated in the GHC 6.4 release in 2005. Further examples and a programmer-oriented tutorial are also available.[19]

Our main war cry is *compositionality*: a programmer can control atomicity and blocking behavior in a modular way that respects abstraction barriers. In contrast, lock-based approaches lead to a direct conflict between abstraction and concurrency (see Section 2). Taken together, these ideas offer a qualitative improvement in language support for modular concurrency, similar to the improvement in moving from assembly code to a high-level language. Just as with assembly code, a programmer with sufficient time and skills may obtain better performance programming directly with low-level concurrency control mechanisms rather than transactions—but for all but the most demanding applications, our higher-level STM abstractions perform quite well enough.

This paper is an abbreviated and polished version of an earlier paper with the same title.[9] Since then there has been a tremendous amount of activity on various aspects of transactional memory, but almost all of it deals with the question of *atomic memory update*, while much less attention is paid to our central concerns of *blocking* and *synchronization* between threads, exemplified by `retry` and `orElse`. In our view this is a serious omission: locks without condition variables would be of limited use.

Transactional memory has tricky semantics, and the original paper gives a precise, formal semantics for transactions, as well as a description of our implementation. Both are omitted here due to space limitations.

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

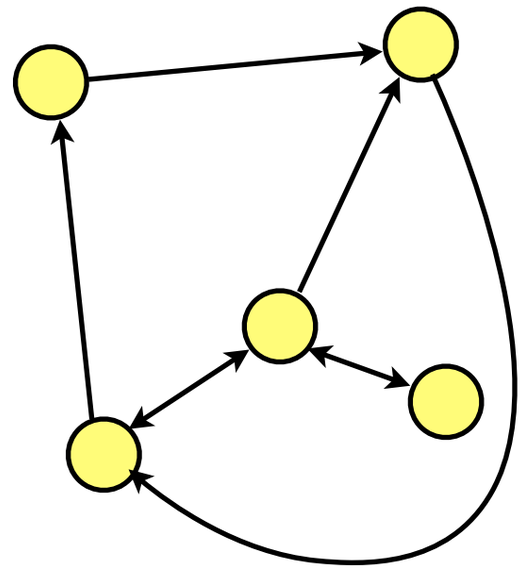
Message Passing Systems

- Actors and message passing
- Immutable data
- Ownership and race conditions

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Message Passing Systems

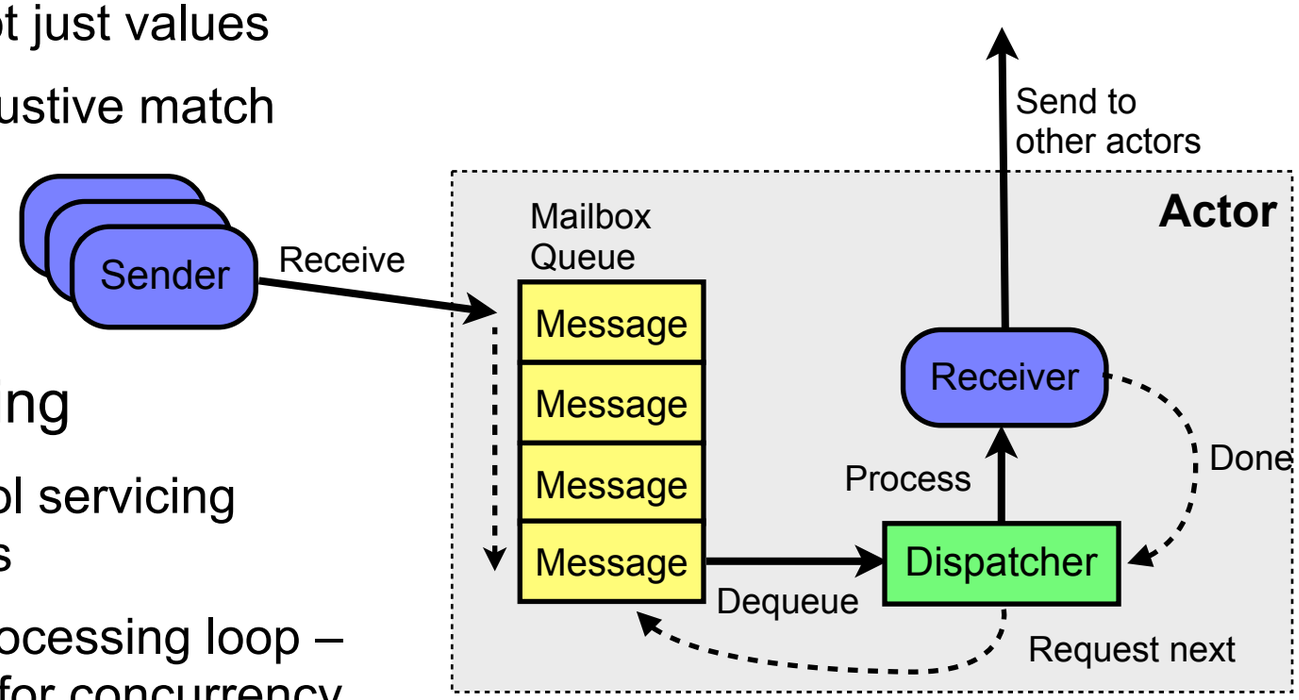
- System is structured as a set of communicating processes, *actors*, with no shared mutable state
- All communication via exchange of messages
  - Messages are generally required to be immutable – data conceptually copied between processes
  - Some systems use linear types to ensure messages are not 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



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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 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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

CC BY ND

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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…
    }
  }
}
```

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

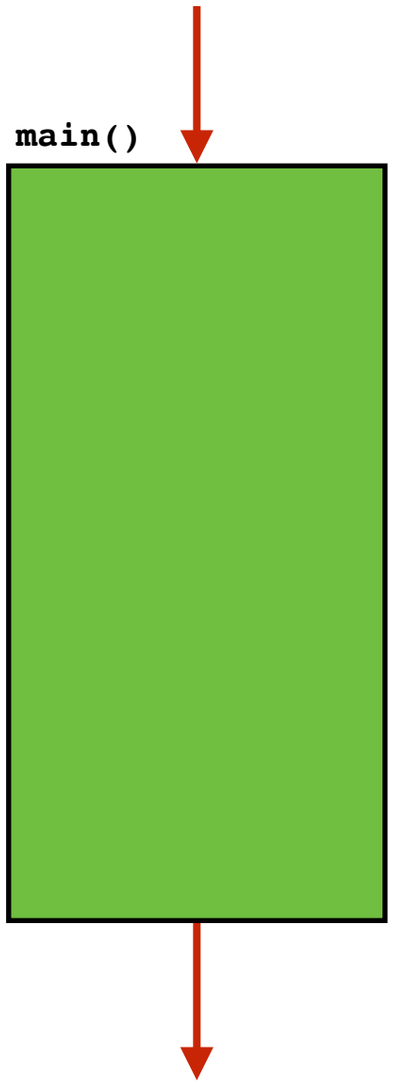
Example: Rust

```
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…
    }
  }
}
```



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

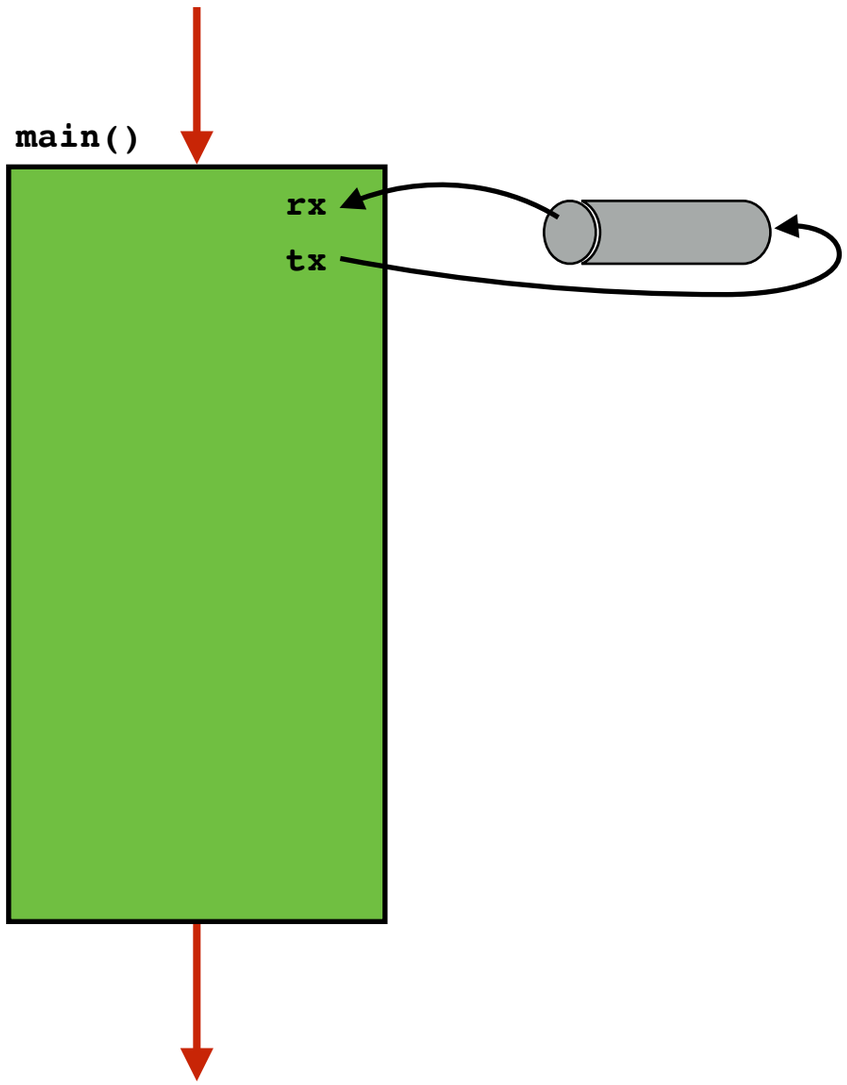
Example: Rust

```
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…
    }
  }
}
```



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

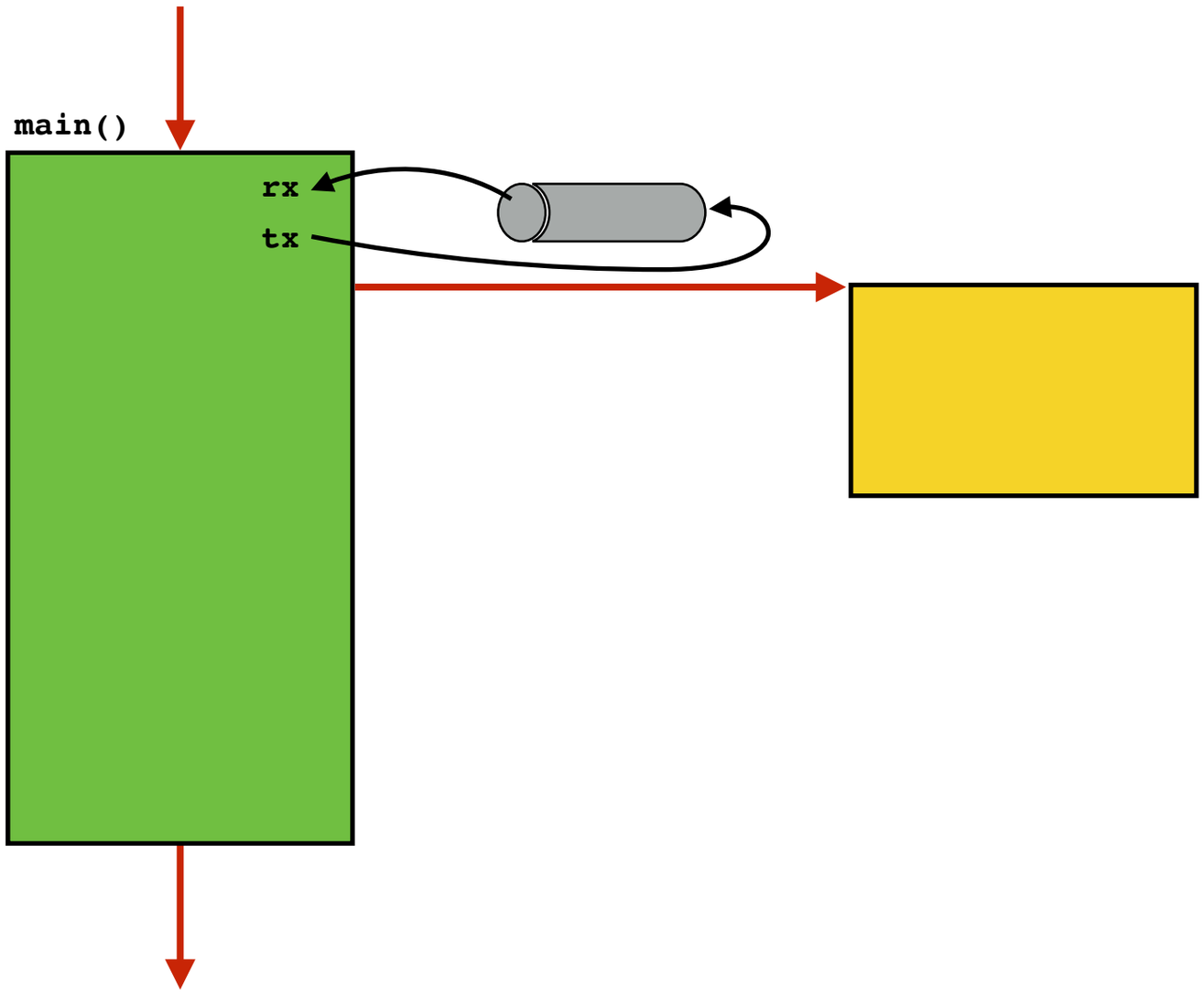
Example: Rust

```
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…
    }
  }
}
```



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

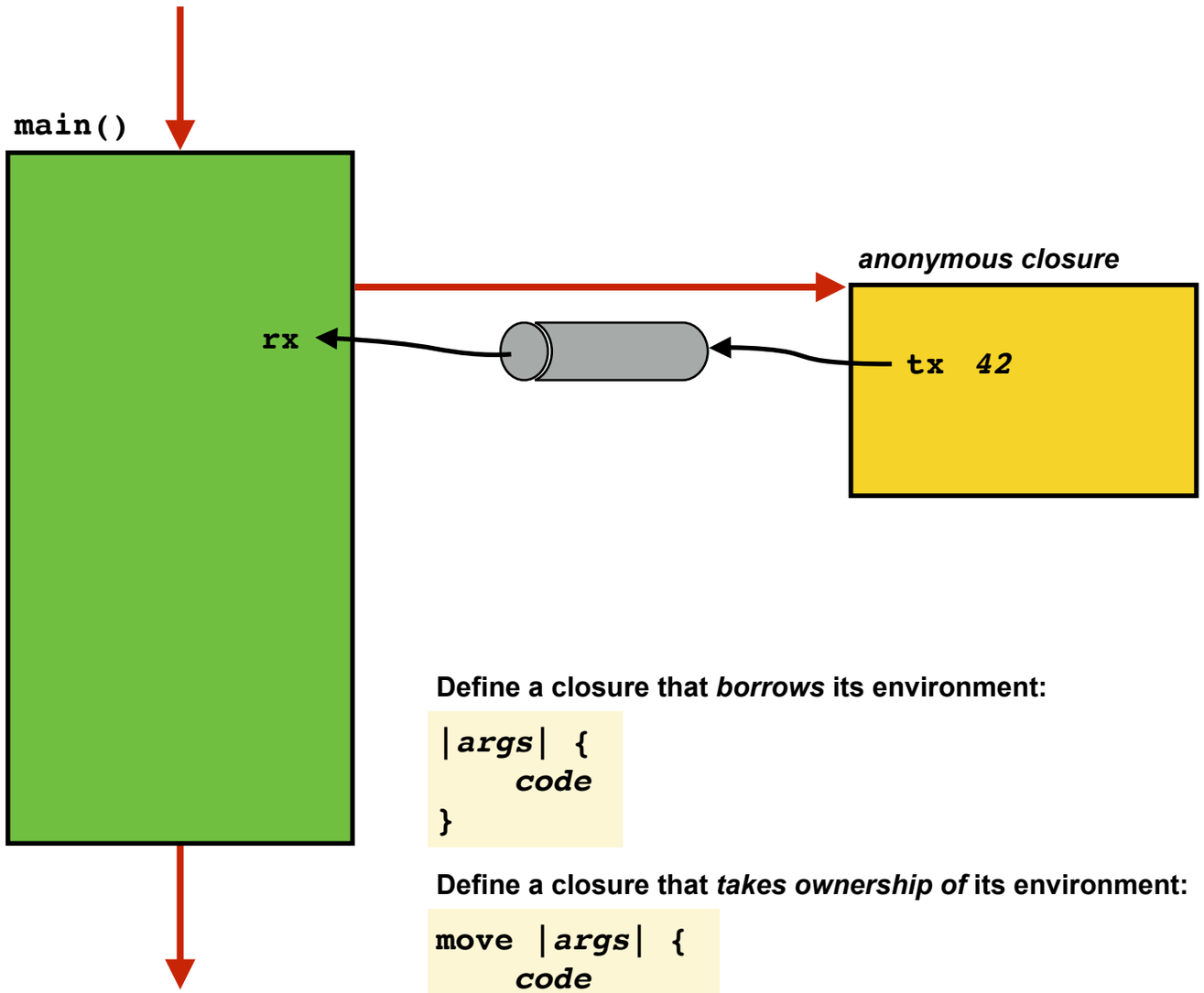
Example: Rust

```
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…
    }
  }
}
```



Define a closure that *borrows* its environment:

```
|args| {
    code
}
```

Define a closure that *takes ownership of* its environment:

```
move |args| {
    code
}
```

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

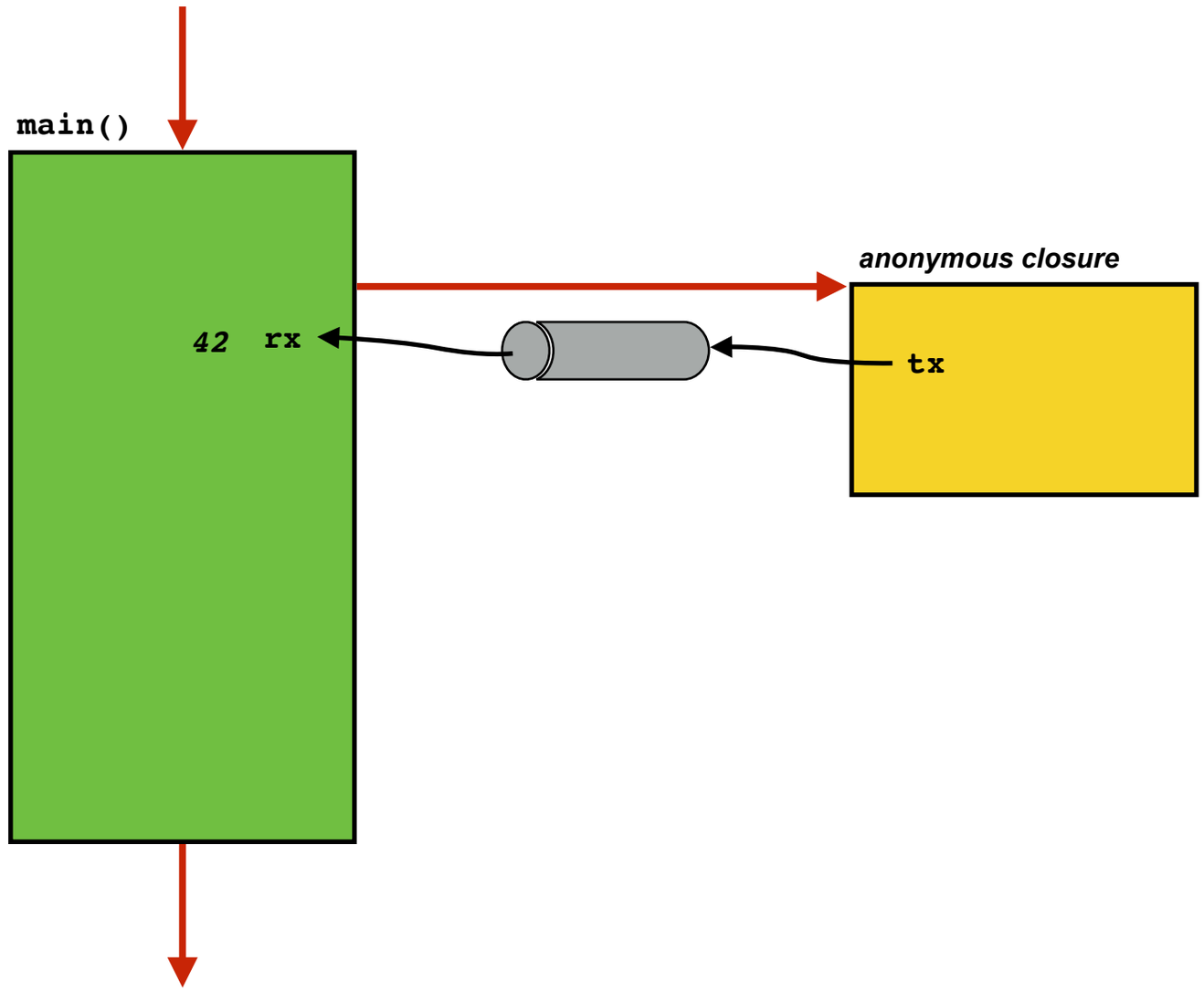
Example: Rust

```
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…
    }
  }
}
```



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Trade-offs

- The two approaches behave quite differently:
  - **Scala+Akka** let 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** uses 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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

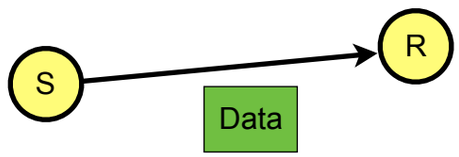
Race Conditions

- What is a race condition?
- Avoidance using immutable data
- Avoidance by controlling ownership

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Race Conditions

- Runtime ensures receiver processes messages sequentially, but it is part of a concurrent system
- Data conceptually copied from sender to receiver
  - In practice, a **reference** to the data is often copied, for performance reasons



- A **race condition** occurs if the data is modified after it is sent, and the modification is visible to the receiver
  - Unclear if the receiver sees the old or new version, depending on timing of changes, scheduling, etc.
  - Leads to unpredictable bugs
- Two approaches to avoid:
  - Immutable data
  - Ownership tracking and transfer

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Avoiding Race Conditions: Immutable Data

- Race conditions cannot occur if data is not modified
  - Ensure any objects to be sent between threads is immutable
  - Erlang ensures this in the language → *all* variables are immutable
  - Scala+Akka requires programmer discipline → potential race conditions if message data modified after message sent

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Avoiding Race Conditions: Ownership Transfer

- Race conditions cannot occur if data is not shared
  - Ensure ownership of an object is transferred, so the sender cannot access the object after it has been sent
  - Natural fit for Rust:
    - Standard library provides a **channel** abstraction
    - The **`send()`** function takes ownership of the data to be sent
    - The **`recv()`** function returns ownership of the received data
    - The usual ownership rules in the type system ensure the data is not accessible once it has been sent
- If sender cannot access the object once it has been sent, it cannot change it → race condition prevented

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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();
}

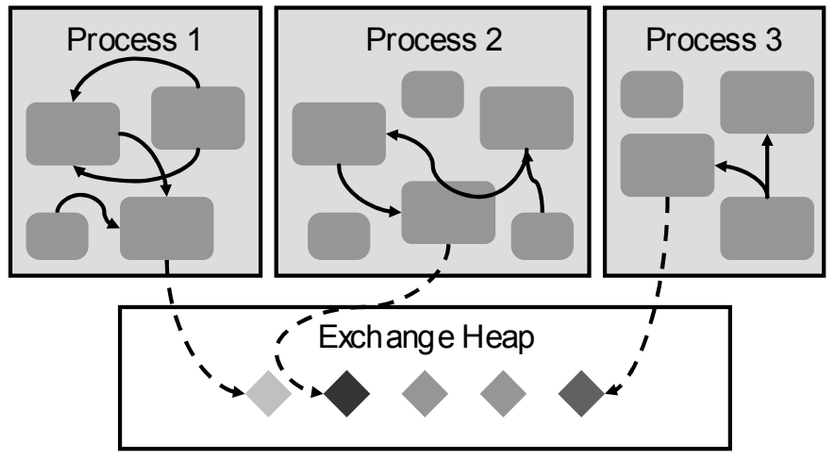
% rustc test.rs
test.rs:15:5: 15:18 error: use of moved value: `message` [E0382]
test.rs:15     message.x = 6;
               ^~~~~~~~~~~~~
```

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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



[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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Summary

- Concurrency and memory models
- Transactions
- Message Passing

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow