Lecture Outline

• Concurrency, threads, and locks
• Limitations of lock-based concurrency
  • Memory models
  • Composition and correctness
• Message passing systems
  • Approaches and principles
  • Erlang
  • Scala+Akka
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
  - Generally enforce synchronisation via explicit locks around critical sections (e.g. Java synchronized methods and statements; pthread mutexes)
  - Very 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
Multicore Memory Models

- Memory typically shared between cores
  - May be symmetric or NUMA; potentially multiple layers of caching
- When do writes made by one core become visible to other cores?
  - 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
  - Varies between processor architectures – differences generally hidden by language runtime, provided language has a clear memory model
Multicore Memory Models

- Memory Model defines space in which language runtime and processor architecture can innovate, without breaking programs.
  - Synchronisation between threads occurs only at well-defined instants; memory may appear inconsistent between these times, if that helps the processor and/or runtime system performance.
  - Without well-defined memory model, cannot reason about lock-based code.
  - Essential for portable code using locks and shared memory.
Example: Java Memory Model

• Java has a formally defined memory model

• Between 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 the field is `volatile`, or if a synchronisation lock is held

• Within a thread: actions are seen in program order

[Somewhat simplified: see Java Language Specification, Chapter 17, for details http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf]
Multicore Memory Models

• 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++, in particular, have very poorly specified memory models
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
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?
  • Message passing systems and actor-based languages
  • Transactional memory coupled with functional languages (e.g., Haskell) for automatic rollback and retry of transactions
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
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 via the 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
Message Passing Systems

• Message passing starting to see wide deployment
  • Erlang (http://www.erlang.org/)
  • Scala (http://www.scala-lang.org/) + Akka (http://akka.io/)
  • Both adopt a similar message passing model:
    • Asynchronous – messages are buffered at receiver; sender does not wait
    • 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

• Other systems make different design choices
  • Singularity (discussed in Tutorial 3) and the Rust programming language (http://rust-lang.org/) use asynchronous statically typed messages passed via explicit channels
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"
}
Advantages and Disadvantages

• Model adopted by Erlang and Scala+Akka gives weakly coupled processes that 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

• Disadvantage: checking happens at run time, so guarantees of robustness are probabilistic
  • Statically typed message passing provides compile-time checking that a process can respond to messages
  • Rendezvous-based synchronous systems provide better tests for liveness
Further Reading


- Does the programming model make sense?
- Does the reliability model (“let it crash”) make sense? Will discuss further in next lecture