Notes on Threads and in-box Concurrency

matt@cs.uoregon.edu

Topics

- Java threading
- Erlang

Java Concurrency

- First, Java.
- Java has two routes to concurrency, although one is much easier to use than the other.
    - Threads. (The easy one)
    - Remote Method Invocation. (RMI)

Java Threads

- Threading in Java is seen in two distinct parts of the language:
  - In the language itself in the form of keywords.
    - Keywords are available in Java such as **`volatile`** and **`synchronized`** that are used to control data visibility and critical section lock management.
  - In the standard library.
    - **`java.util.concurrent.*`**
    - **`java.lang.Thread`**

Threads

- Thread-based code in Java is most often implemented in one of two ways:
  - Extending **`java.lang.Thread`** and implementing the **`run()`** method to represent the body of the thread.
  - Implementing the **`java.lang.Runnable`** interface and handing the class to a thread wrapper that actually executes it.
- Both are similar (in fact, Thread implements Runnable). Runnable is a nice abstraction to work with since many thread scheduling and resource pooling classes exist that take runnable objects as arguments.
  - Extending thread is, to my knowledge, most useful if you want to build fairly complex thread-based objects with state and possibly overriding some of the primitive thread operations. Runnable is good if you only care about run().

java.lang.Thread

- This is the basic thread object. It has a pretty detailed API of functions related to the Thread, beyond simply the thread body implemented in run().
- You can interrupt threads, set and query scheduling priorities, yield to other threads, and sleep.
- What is interesting is that a large portion of the API that was heavily used in early versions of Java are now deprecated because they provide far too much opportunity to shoot yourself in the foot (they practically encourage you to do it). These are start(), stop(), suspend(), resume(), and others.
- The mass deprecation of functions makes me wonder if the Thread versus Runnable separation is a legacy of the original design, but would possibly be different in light of this late change to the API.

`java.lang.Runnable`

- Runnable is simple - it is an interface with a single function called "`run()`" that must be implemented.

- `run()` is the thread body. Think of it as similar to the function pointer that `pthread_create` is given to spawn off a pthreads thread, or the function handed to the Erlang `spawn` function.

- A runnable object is run inside of a thread by either creating a thread with the runnable object as a constructor argument, or by handing the runnable object to a pooling or execution management class that spawns threads under the covers for you.

Keywords

- The language defines two of keywords that are related to synchronization and memory consistency.

`volatile`

- The **`volatile`** keyword on a variable indicates that it is shared. This tells the compiler to:
  - Restrict how memory operations are reordered with respect to the **`volatile`** variable.
  - Ensure that the data is not cached in a register or cache such that any processor that reads the data will get the most recent value.

`synchronized`

- A `synchronized` block is used to enforce atomicity.
- `Synchronized` methods are equivalent to a `synchronized` block that contains the entire method body.
- `Synchronized` blocks are made up of two parts: the lock object and the block during which the lock must be held to execute.

```
synchronized (obj) {
  // some code
}
```

`synchronized`

- A **`synchronized`** block is used to enforce atomicity.
- **`Synchronized`** methods are equivalent to a **`synchronized`** block that contains the entire method body.
- **`Synchronized`** blocks are made up of two parts: the lock object and the block during which the lock must be held to execute.

Lock object

```
synchronized (obj) {
  // some code
}
```

`synchronized`

- A `synchronized` block is used to enforce atomicity.
- `Synchronized` methods are equivalent to a `synchronized` block that contains the entire method body.
- `Synchronized` blocks are made up of two parts: the lock object and the block during which the lock must be held to execute.

```
synchronized (obj) {
  // some code
}
```

Body that executes only after the lock has been acquired. The lock is released when the block is exited.

Methods

- You’ve probably encountered this more frequently than you have with synchronized blocks.
- A synchronized method simply means that the entire method is a synchronized block. Unfortunately, the lock object is associated with the class instance (the **`this`** object of the callee) containing the method.
- What does this mean?
    - All methods that are synchronized will share a lock. This can have a horrible impact on performance.

Example: Synchronized methods

```
public synchronized void method1() {
    for (int i = 0; i < 1000000; i++) {
        for (int j = 0; j < 2000; j++) {
            double x = 1.4 * j;
        }
    }
    System.err.println("METHOD 1!");
}

public synchronized void method2() {
    for (int i = 0; i < 1000000; i++) {
        for (int j = 0; j < 30; j++) {
            double x = 1.4 * j;
        }
    }
    System.err.println("METHOD 2!");
}
```

Example: Synchronized blocks

```
private Object something = new Object();
private Object somethingElse = new Object();

public void method3() {
    synchronized (something) {
        for (int i = 0; i < 1000000; i++) {
            for (int j = 0; j < 2000; j++) { double x = 1.4 * j; }
        }
    }
    System.err.println("METHOD 3!");
}

public void method4() {
    synchronized (somethingElse) {
        for (int i = 0; i < 1000000; i++) {
            for (int j = 0; j < 30; j++) { double x = 1.4 * j; }
        }
    }
    System.err.println("METHOD 4!");
}
```

One versus the other

- This shouldn't be surprising. In the first case, the slow method acquires the lock and the fast method gets stuck waiting for it. In the second case, the two methods use independent lock objects, so the fast methods finishes first while the slow one trails behind.

```
Starting.
METHOD 1!
METHOD 2!
Next.
METHOD 4!
METHOD 3!
```

Intrinsic locks

- What is interesting here? Every object has an *intrinsic lock*.
- This means there is a lock somewhere for every object. This is why one can use any object as the lock object in the synchronized block.
- This also explains why the synchronized methods prevent reentrance into a class from external callers. The lock object is **`this`** on the callee side.
- You should recognize the whole-object locking model for synchronized methods, where a single lock is encapsulated in a class with locking managed by the functions on the object.
    - Can you remember what this was?

Monitors

- It’s the monitor concept.
- The monitor terminology comes up quite frequently in Java literature.
- You can tell the Java team was inspired by the monitor idea.

Reentrant intrinsic locks

- Intrinsic locks are reentrant. What does this mean?
- If a thread holds a lock, and calls a routine that attempts to acquire the same lock, it will do so.
- This becomes especially apparent when dealing with synchronized methods where **`this`** is the lock. What if a derived class calls a synchronized method on a parent class that is also synchronized?
  - If locks were not reentrant, this would deadlock.

Immutable objects and sharing

- Sometimes you wish to share data that isn't mutable between threads.
- The **`final`** keyword can be used to explicitly state (and enforce) this immutability.
- This is good practice - if you want to build code that has shared data that you know will not require locking, **`final`** will ensure that it is in fact not modified by some reckless user.
- It's not clear if the compiler will do any optimizations for concurrent code in the case of **`final`** fields though.

Standard library

- So that’s the Java syntax for concurrency. It’s not much, but it’s more than most languages.
- The standard library contains the rest of the necessary building blocks for building concurrent code.

`java.util.concurrent`

- This contains most of the utility classes for concurrent programming.
- There are two important subpackages:
  - **`java.util.concurrent.atomic`** : These are objects that implement basic data types supporting atomic operations on them. For example, **`AtomicInteger`**.
  - **`java.util.concurrent.locks`** : Custom locking. When the **`synchronized`** keyword just won't do, you can play with the locks yourself.

Important utility classes

- The utility classes in **`java.util.concurrent`** can be broken into a few different types.
    - Executors: These classes handle executing **`Runnable`** classes that are handed to them.
    - Exchanger: A synchronization object that allows two threads to reach some synchronization point at which time they exchange some data.
    - Thread safe data structures
    - ThreadPool management.
    - Synchronization structures

Executors

- An executor is an object that executes runnable objects.
- It hides the thread management from the source of the runnable objects.
- For example, **`ThreadPoolExecutor`** manages a thread pool such that the producer of runnable objects need not worry about how many threads are optimal to have executing at a given time.
- This is quite useful if you have an algorithm that will produce many threads, but you don't want to manually tune it to throttle back from creating too many.
- You can build custom executors by implementing the **`Executor`** interface.

Data structures

- Often one will want thread-safe data structures available in a multithreaded program. For example, a shared queue containing tasks to be performed that guarantees atomicity of push and pop.
- There are a few different Queues and a HashMap. Read the Java API docs for specifics on these. They're not too difficult, but not really worth covering in detail here.
- One interesting case is the CopyOnWriteArrayList and Set. These are structures where a write forces a brand new copy of the underlying data structure to be made atomically. Sounds expensive, eh?
    - Sort of. These exist if reads vastly outnumber writes. In that case, why pay the penalty for synchronized accessor functions if the likelihood of a write occurring and causing a problem is low?

Synchronization structures

- Finally, we also have the obligatory synchronization structures.
  - Semaphore: Does what it says.
  - CyclicBarrier: This is a typical barrier.
    - For some reason, the Java people seemed to think that the name “Barrier” alone would imply a one-shot-only structure instead of the typical case where the Barrier resets when it is used.
    - So, Cyclic means it resets and can be used over and over. This would be worth it if Java implemented a non-cyclic barrier... But it doesn’t.

Erlang: A real world concurrent language

- Erlang.
- Invented at Ericsson telecommunications approx. 20 years ago for the purpose of writing code for telecommunications systems.
    - Distributed systems: interacting telecom switches.
    - Robust to failures.
    - Tackling intrinsically parallel problems.

Erlang: Right place, right time.

- Erlang is getting a great deal of attention because it happened to find itself at the right place at the right time.
- **Right place**: It is a very mature language with a large user base that is already concerned with efficient use of parallel processing resources.
- **Right time**: Multicore is here, and we need a language to target it!

So, what is Erlang?

- Erlang is a single assignment declarative language.
  - Derived from Prolog.
- Provides ML-like pattern matching facilities for defining functions.
- Syntactic means provided to create threads and communicate between them.
- Built to withstand reliability problems with distributed systems and hot-swapping of code.
- Concurrency is based on the “actor model”.

Using Erlang.

- Let’s do something simple. A routine that squares numbers.

```erlang
-module(squarer).
-export([square/1]).

square(N) -> N*N.
```

Using Erlang.

- Let’s do something simple. A routine that squares numbers.

Define a module
to hold the code.

```
-module(squarer).
-export([square/1]).

square(N) -> N*N.
```

Using Erlang.

- Let’s do something simple. A routine that squares numbers.

```
-module(squarer).
-export([square/1]).

square(N) -> N*N.
```

Export the squaring function which has arity 1.

Using Erlang.

- Let’s do something simple. A routine that squares numbers.

```
-module(squarer).
-export([square/1]).

square(N) -> N*N.
```

Define the function.

Interactive session

```
matt@magnolia [code]$ erl
Erlang (BEAM) emulator version 5.6.1 [source] [smp:2]
[async-threads:0] [kernel-poll:false]

Eshell V5.6.1  (abort with ^G)
1> c(squarer).
{ok,squarer}
2> squarer:square(2).
4
3> squarer:square(7654).
58583716
4>
```

Interactive session

Start interpreter.

```
matt@magnolia [code]$ erl
Erlang (BEAM) emulator version 5.6.1 [source] [smp:2]
[async-threads:0] [kernel-poll:false]

Eshell V5.6.1  (abort with ^G)
1> c(squarer).
{ok,squarer}
2> squarer:square(2).
4
3> squarer:square(7654).
58583716
4>
```

Interactive session

```
matt@magnolia [code]$ erl
Erlang (BEAM) emulator version 5.6.1 [source] [smp:2]
[async-threads:0] [kernel-poll:false]

Eshell V5.6.1  (abort with ^G)
1> c(squarer).
{ok,squarer}
2> squarer:square(2).
4
3> squarer:square(7654).
58583716
4>
```

Compile module in squarer.erl.

Interactive session

```
matt@magnolia [code]$ erl
Erlang (BEAM) emulator version 5.6.1 [source] [smp:2]
[async-threads:0] [kernel-poll:false]

Eshell V5.6.1  (abort with ^G)
1> c(squarer).
{ok,squarer}
2> squarer:square(2).
4
3> squarer:square(7654).
58583716
4>
```

Result.

Interactive session

```
matt@magnolia [code]$ erl
Erlang (BEAM) emulator version 5.6.1 [source] [smp:2]
[async-threads:0] [kernel-poll:false]

Eshell V5.6.1  (abort with ^G)
1> c(squarer).
{ok,squarer}
2> squarer:square(2).
4
3> squarer:square(7654).
58583716
4>
```

Call function.

Interactive session

```
matt@magnolia [code]$ erl
Erlang (BEAM) emulator version 5.6.1 [source] [smp:2]
[async-threads:0] [kernel-poll:false]

Eshell V5.6.1  (abort with ^G)
1> c(squarer).
{ok,squarer}
2> squarer:square(2).
4
3> squarer:square(7654).
58583716
4>
```

Result.

Interactive session

```
matt@magnolia [code]$ erl
Erlang (BEAM) emulator version 5.6.1 [source] [smp:2]
[async-threads:0] [kernel-poll:false]

Eshell V5.6.1  (abort with ^G)
1> c(squarer).
{ok,squarer}
2> squarer:square(2).
4
3> squarer:square(7654).
58583716
4>
```

Call function.

Interactive session

```
matt@magnolia [code]$ erl
Erlang (BEAM) emulator version 5.6.1 [source] [smp:2]
[async-threads:0] [kernel-poll:false]

Eshell V5.6.1  (abort with ^G)
1> c(squarer).
{ok,squarer}
2> squarer:square(2).
4
3> squarer:square(7654).
58583716
4>
```

Result.

Interactive session

```
matt@magnolia [code]$ erl
Erlang (BEAM) emulator version 5.6.1 [source] [smp:2]
[async-threads:0] [kernel-poll:false]

Eshell V5.6.1  (abort with ^G)
1> c(squarer).
{ok,squarer}
2> squarer:square(2).
4
3> squarer:square(7654).
58583716
4>
```

Lists

- Things get more interesting when we start using lists, one of the primary functional programming data structures.

- How about a function to square every element in a list?

```
squarelist([H|T]) -> [square(H)|squarelist(T)];
squarelist([])    -> [].
```

- This uses **pattern matching** on the argument to determine which body to execute. The list is matched as either empty [], or as a **H**ead atom followed by a **T**ail list. Recursion used to walk down the list until nothing left (empty).

Lists

squarer.erl

```erlang
-module(squarer).
-export([square/1, squarelist/1]).

square(N) -> N*N.

squarelist([H|T]) -> [square(H)|squarelist(T)];
squarelist([])    -> [].
```

```
1> c(squarer).
{ok,squarer}
2> L=[1,2,3,4].
[1,2,3,4]
3> L2=squarer:squarelist(L).
[1,4,9,16]
4> squarer:squarelist(L2).
[1,16,81,256]
```

Quicksort

- This is an example of a list-comprehension based algorithm for implementing quicksort. It isn't the most efficient algorithm - it's presented more or less as a demonstration of elegant syntax.

```
% quicksort from Programming Erlang
-module(sort).
-export([sort/1]).

sort([]) -> [];
sort([Pivot|T]) ->
  sort([X || X <- T, X < Pivot])
  ++ [Pivot] ++
  sort([X || X <- T, X >= Pivot]).
```

Looking closer

- How is this working?

```
1> L=[13,17,5,3,9,1,2,6,3,22].
[13,17,5,3,9,1,2,6,3,22]
2> [Pivot|T] = L.
[13,17,5,3,9,1,2,6,3,22]
3> [X || X <- L, X < Pivot].
[5,3,9,1,2,6,3]
4> [X || X <- L, X >= Pivot].
[13,17,22]
```

Looking closer

- How is this working?

```
1> L=[13,17,5,3,9,1,2,6,3,22].
[13,17,5,3,9,1,2,6,3,22]
2> [Pivot|T] = L.
[13,17,5,3,9,1,2,6,3,22]
3> [X || X <- L, X < Pivot].
[5,3,9,1,2,6,3]
4> [X || X <- L, X >= Pivot].
[13,17,22]
```

Create a list.

Looking closer

- How is this working?

Separate into
pivot and tail.

```
1> L=[13,17,5,3,9,1,2,6,3,22].
[13,17,5,3,9,1,2,6,3,22]
2> [Pivot|T] = L.
[13,17,5,3,9,1,2,6,3,22]
3> [X || X <- L, X < Pivot].
[5,3,9,1,2,6,3]
4> [X || X <- L, X >= Pivot].
[13,17,22]
```

Looking closer

- How is this working?

```
1> L=[13,17,5,3,9,1,2,6,3,22].
[13,17,5,3,9,1,2,6,3,22]
2> [Pivot|T] = L.
[13,17,5,3,9,1,2,6,3,22]
3> [X || X <- L, X < Pivot].
[5,3,9,1,2,6,3]
4> [X || X <- L, X >= Pivot].
[13,17,22]
```

List comprehension to create a list of elements less than the pivot.

Looking closer

- How is this working?

```
1> L=[13,17,5,3,9,1,2,6,3,22].
[13,17,5,3,9,1,2,6,3,22]
2> [Pivot|T] = L.
[13,17,5,3,9,1,2,6,3,22]
3> [X || X <- L, X < Pivot].
[5,3,9,1,2,6,3]
4> [X || X <- L, X >= Pivot].
[13,17,22]
```

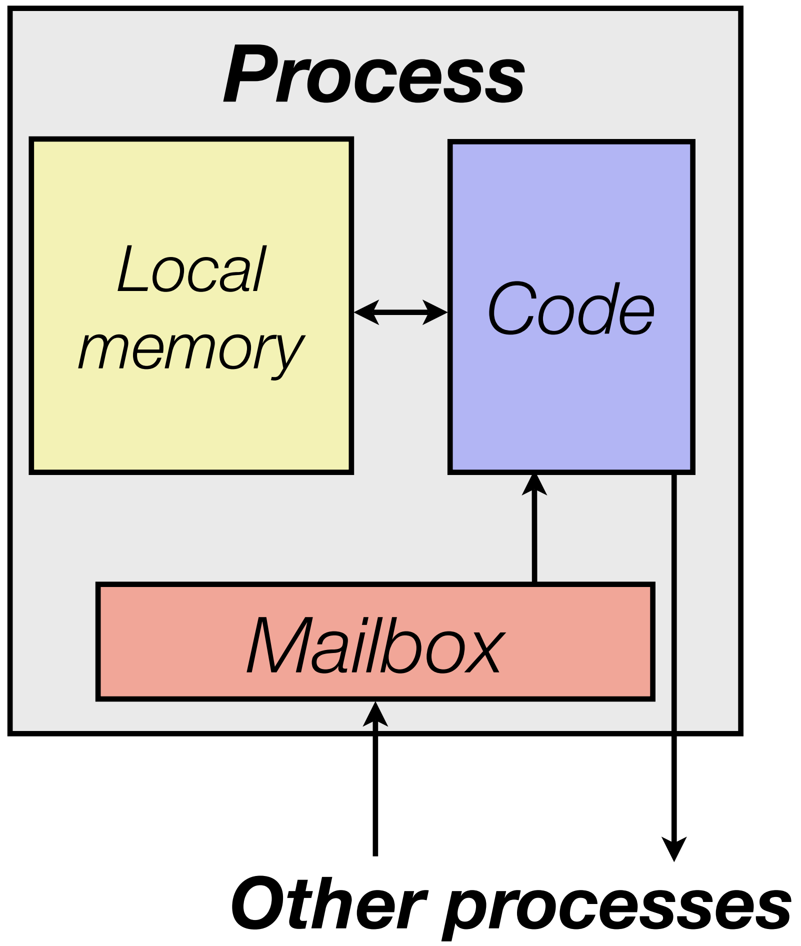
List comprehension to create a list of elements greater than or equal to the pivot.

Erlang for concurrency

- What really makes Erlang interesting is it’s application to concurrency. Otherwise, it’s just another declarative language (although, a pretty nice one).
- Erlang has gained a great deal of attention lately because it promises hope for one of the big problems in computing right now:
    - Concurrency is important.
    - Writing **correct** and **efficient** code is hard. Productivity is bad when it comes to writing good concurrent code.
    - Erlang was designed to fix this.

Actor model

- We covered this earlier in the semester briefly. The basic idea is:
    - Non-shared memory process (ie: each process has thread-local memory only).
    - Each process has a mailbox into which it can receive messages.
    - Each process has a unique identifier.
- Note that Erlang uses the term **process** to mean a very light-weight process that is essentially a thread with a miniscule amount of extra state and functionality related to mailboxes and bookkeeping.



Message passing syntax

- Concurrency is implemented by creating a set of processes, and letting them communicate with each other via messages.
- Processes, by definition, run concurrently. When multiple processing elements are available, such as in an SMP or Multicore, the processes run in parallel.
- First, when processes are created, they are given unique process identifiers (or PIDs).
- Messages are sent by passing tuples to a PID with the `!` syntax.
  - `PID ! {message}.`
- Messages are retrieved from the mailbox using the `receive()` function.

A simple example

- Let’s write a parallel Fibonacci function that will run on two cores.
- The idea:
    - Fib(N) = Fib(N-1) + Fib(N-2).
    - Compute each part of the sum on a separate core. Note that we will NOT make a thread for each recursive step - just the first one.

Parallel Fibonacci numbers

```
% test to compute fibonacci numbers
-module(fib).
-export([start/0, fib/1, sfib/1]).

start() -> spawn(fun loop/0).

fib(N) ->
  Pid = start(),
  Qid = start(),
  Pid ! {self(), N-1},
  Qid ! {self(), N-2},
  receive
    {_, ResponseP} ->
      receive
        {_, ResponseQ} -> ResponseP+ResponseQ
      end
  end.
```

```
sfib(0) -> 1;
sfib(1) -> 1;
sfib(N) -> sfib(N-1)+sfib(N-2).

loop() ->
  receive
    {From, N} ->
      From ! {self(), sfib(N)},
      loop()
  end.
```

Parallel Fibonacci numbers

```
% test to compute fibonacci numbers
-module(fib).
-export([start/0, fib/1, sfib/1]).

start() -> spawn(fun loop/0).

fib(N) ->
  Pid = start(),
  Qid = start(),
  Pid ! {self(), N-1},
  Qid ! {self(), N-2},
  receive
    {_, ResponseP} ->
      receive
        {_, ResponseQ} -> ResponseP+ResponseQ
      end
  end.
```

```
sfib(0) -> 1;
sfib(1) -> 1;
sfib(N) -> sfib(N-1)+sfib(N-2).

loop() ->
  receive
    {From, N} ->
      From ! {self(), sfib(N)},
      loop()
  end.
```

Function to create new threads with loop() as their body.

Parallel Fibonacci numbers

```
% test to compute fibonacci numbers
-module(fib).
-export([start/0, fib/1, sfib/1]).

start() -> spawn(fun loop/0).

fib(N) ->
  Pid = start(),
  Qid = start(),
  Pid ! {self(), N-1},
  Qid ! {self(), N-2},
  receive
    {_, ResponseP} ->
      receive
        {_, ResponseQ} -> ResponseP+ResponseQ
      end
  end.
```

```
sfib(0) -> 1;
sfib(1) -> 1;
sfib(N) -> sfib(N-1)+sfib(N-2).
```

```
loop() ->
  receive
    {From, N} ->
      From ! {self(), sfib(N)},
      loop()
  end.
```

Loop function that blocks on receive until a tuple is received. At that time it does some work, responds with the answer, and repeats. Note tail recursion.

Parallel Fibonacci numbers

```
% test to compute fibonacci numbers
-module(fib).
-export([start/0, fib/1, sfib/1]).

start() -> spawn(fun loop/0).

fib(N) ->
  Pid = start(),
  Qid = start(),
  Pid ! {self(), N-1},
  Qid ! {self(), N-2},
  receive
    {_, ResponseP} ->
      receive
        {_, ResponseQ} -> ResponseP+ResponseQ
      end
  end.
```

```
sfib(0) -> 1;
sfib(1) -> 1;
sfib(N) -> sfib(N-1)+sfib(N-2).

loop() ->
  receive
    {From, N} ->
      From ! {self(), sfib(N)},
      loop()
  end.
```

Fib() function that creates two threads, sends subproblems to each, and collects the result back up.

`start/0`

- `start = spawn(fun loop/0).`
- This function spawns a process with the loop function as it’s body, and returns the Pid of the process (which is the return value of spawn).

loop/0

```
loop() ->
  receive
    {From, N} ->
      From ! {self(), sfib(N)},
      loop()
  end.
```

`loop/0`

Blocking receive until a message comes in.

```
loop() ->
  receive
    {From, N} ->
      From ! {self(), sfib(N)},
      loop()
  end.
```

loop/0

```
loop() ->
  receive
    {From, N} ->
      From ! {self(), sfib(N)},
      loop()
  end.
```

Match messages that look like {From, N}. From is the PID of the message source, N is the data.

loop/0

```
loop() ->
  receive
    {From, N} ->
      From ! {self(), sfib(N)},
      loop()
  end.
```

Create a new tuple that has the processes own PID as the first element and the result of a sequential fib() function on N.

loop/0

```
loop() ->
  receive
    {From, N} ->
      From ! {self(), sfib(N)},
      loop()
  end.
```

Send this to the source of the original message.

loop/0

```
loop() ->
  receive
    {From, N} ->
      From ! {self(), sfib(N)},
      loop()
  end.
```

Repeat.

fib/1

```
fib(N) ->
  Pid = start(),
  Qid = start(),
  Pid ! {self(), N-1},
  Qid ! {self(), N-2},
  receive
    {_, ResponseP} ->
      receive
        {_, ResponseQ} -> ResponseP+ResponseQ
      end
  end.
```

fib/1

Spawn two worker processes.

```
fib(N) ->
  Pid = start(),
  Qid = start(),
  Pid ! {self(), N-1},
  Qid ! {self(), N-2},
  receive
    {_, ResponseP} ->
      receive
        {_, ResponseQ} -> ResponseP+ResponseQ
      end
  end.
```

fib/1

```
fib(N) ->
  Pid = start(),
  Qid = start(),
  Pid ! {self(), N-1},
  Qid ! {self(), N-2},
  receive
    {_, ResponseP} ->
      receive
        {_, ResponseQ} -> ResponseP+ResponseQ
      end
  end.
```

Send them the subproblems. Sends do not block, so they execute in parallel.

fib/1

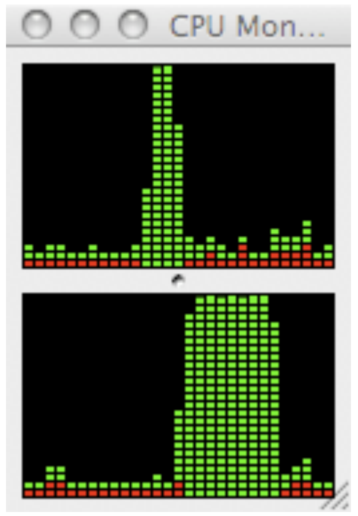
```
fib(N) ->
  Pid = start(),
  Qid = start(),
  Pid ! {self(), N-1},
  Qid ! {self(), N-2},
  receive
    {_, ResponseP} ->
      receive
        {_, ResponseQ} -> ResponseP+ResponseQ
      end
  end.
```

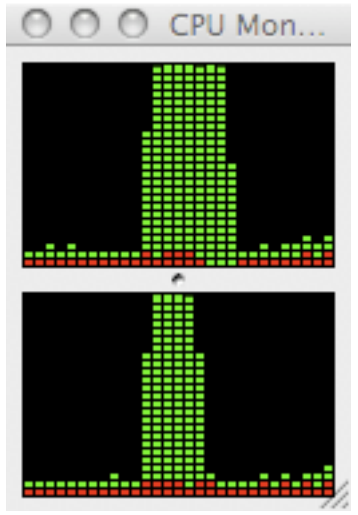
Blocking receive for one message and then the other. Note that generic _ pattern used for source, so we don't care which order they come in.

Is it worth it?

- Do we see a performance benefit? **Yes!** Note that in this case, the computational load is largely the recursion for computing the Fibonacci numbers.
  - Yes, this is probably the worst way to compute these in a practical sense, but it does serve to demonstrate a point.

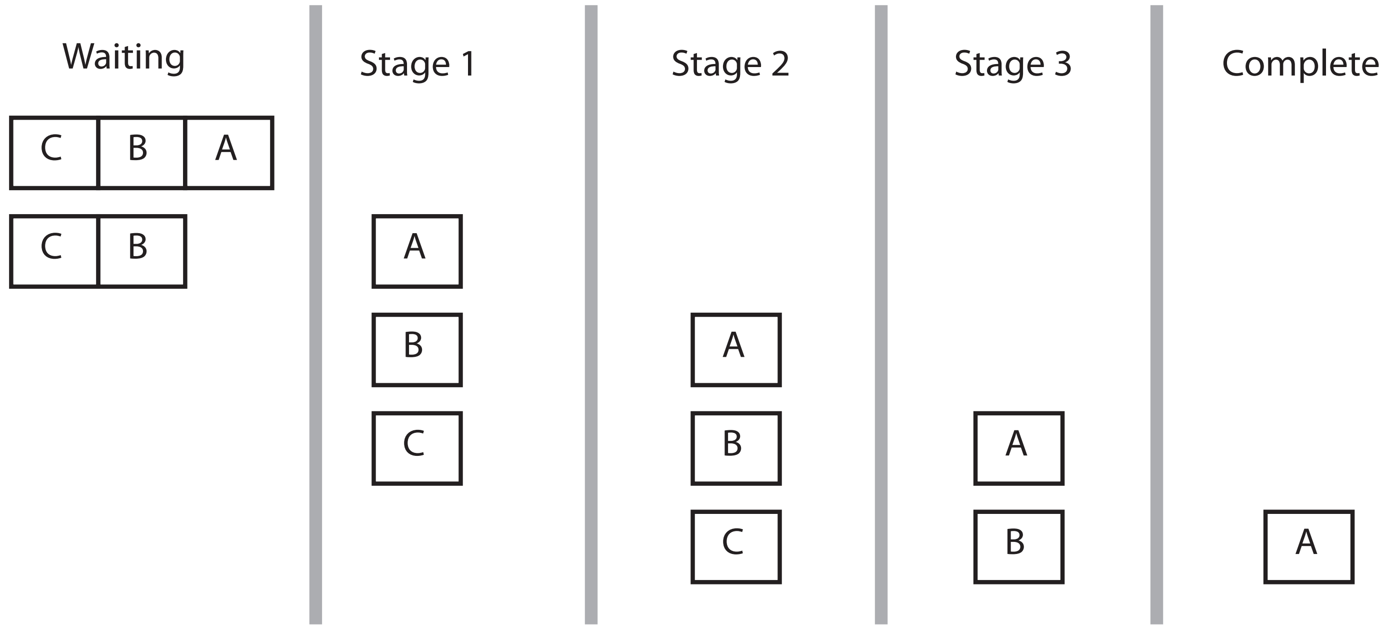
```
1> c(fib).
{ok,fib}
2> fib:sfib(40).
RUNTIME: 12170 / WALLCLOCK: 12203
165580141
3> fib:fib(40).
RUNTIME: 12490 / WALLCLOCK: 8136
165580141
```





Parallel Pipeline Pattern

- The Fibonacci number example is a bit contrived. How about a realistic pattern?
- One that is seen frequently in practice is that of *pipelined parallelism*. This is basically the same idea we saw before in hardware, but applied to software.
- **The idea**: Break a complex problem into a sequence of simpler problems, and send work through the pipeline accumulating up partially complete results until all stages have been reached. When data moves from one stage to the next, work for the next data item can move in behind it and start working.



Process registration

- First, let's introduce the Erlang process registration scheme. The idea here is that a process can register a PID such that other processes can look up the PID by name. Otherwise, PIDs are private, so one cannot access a process that has not either given out it's PID or registered it.

- Four basic functions:

  - **`register(AnAtom, Pid)`** : Associate **`Pid`** with **`AnAtom`**.

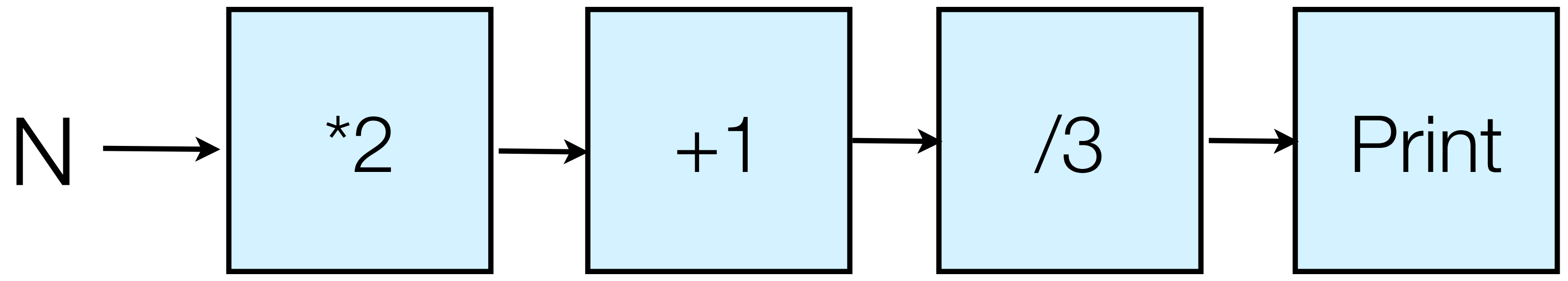
  - **`unregister(AnAtom)`** : Remove entry for **`AnAtom`**.

  - **`whereIs(AnAtom) -> Pid | undefined`** : Look up a **`Pid`** by **`AnAtom`**.

  - **`registered() -> [AnAtom::atom()]`** : Return a list of all registered processes.

Pipeline

- We will implement a pipeline that will compute:
    - print(((N*2)+1) / 3)
- Stages: Doubler -> Adder -> Divider -> Printer
- It’s contrived, but illustrates the basic structure.



Pipeline

```
-module(pipeline).
-export([start/0]).

start() ->
  register(double,spawn(fun loop/0)),
  register(add,spawn(fun loop/0)),
  register(divide,spawn(fun loop/0)),
  register(complete,spawn(fun loop/0)).

% stage 1 : double the input
doubler(N) -> N*2.

% stage 2 : add one to the input
adder(N) -> N+1.

% stage 3 : divide input by three
divider(N) -> N/3.

% stage 4 : consume the results
completer(N) -> io:format("COMPLETE: ~p ~n",[N]).

% receive loop that hands work the the appropriate stage
loop() ->
  receive
    {double, N} -> add ! {add, doubler(N)}, loop();
    {add, N} -> divide ! {divide, adder(N)}, loop();
    {divide, N} -> complete ! {complete, divider(N)}, loop();
    {complete, N} -> completer(N), loop()
  end.
```

Starting up

- The start/0 function fires up the four pipeline stages. Each stage is given a name in the process registry associated with what it does.
- The names are later used by the loop/0 function to figure out sources and destinations of messages.
- A single loop/0 is used for all stages - it simply will invoke the proper stage based on the message structure. More on this in a couple slides.

```
start() ->
  register(double,spawn(fun loop/0)),
  register(add,spawn(fun loop/0)),
  register(divide,spawn(fun loop/0)),
  register(complete,spawn(fun loop/0)).
```

Stage definitions

- Each stage is defined as a standalone function.
- The stage functions themselves know nothing about their context in the pipelined program. They can function just fine as standalone functions.

```
% stage 1 : double the input
doubler(N) -> N*2.

% stage 2 : add one to the input
adder(N) -> N+1.

% stage 3 : divide input by three
divider(N) -> N/3.

% stage 4 : consume the results
completer(N) -> io:format("COMPLETE: ~p ~n",[N]).
```

Message receipt and routing

- The shared loop/0 function iteratively receives messages. First, it matches on the destination (the first element of the tuples). The second argument for each is the message payload, which is just N.

- Each clause in the receive invokes the appropriate stage, and packages it up into a message containing the name of the next stage, and then sends it to the registered name of the next process handing the appropriate stage.

- Note that although the loop/0 function is shared for each stage, independent instances of it exist for each.

```
% receive loop that hands work the the appropriate stage
loop() ->
  receive
    {double, N}   -> add ! {add, doubler(N)}, loop();
    {add, N}      -> divide ! {divide, adder(N)}, loop();
    {divide, N}   -> complete ! {complete, divider(N)}, loop();
    {complete, N} -> completer(N), loop()
  end.
```

Message receipt and routing

- The shared loop/0 function iteratively receives messages. First, it matches on the destination (the first element of the tuples). The second argument for each is the message payload, which is just N.
- Each clause in the receive invokes the appropriate stage, and packages it up into a message containing the name of the next stage, and then sends it to the registered name of the next process handing the appropriate stage.
- Note that although the loop/0 function is shared for each stage, independent instances of it exist for each.

```
% receive loop that hands work the the appropriate stage
loop() ->
  receive
    {double, N}   -> add ! {add, doubler(N)}, loop();
    {add, N}      -> divide ! {divide, adder(N)}, loop();
    {divide, N}   -> complete ! {complete, divider(N)}, loop();
    {complete, N} -> completer(N), loop()
  end.
```

Message receipt and routing

- The shared loop/0 function iteratively receives messages. First, it matches on the destination (the first element of the tuples). The second argument for each is the message payload, which is just N.

- Each clause in the receive invokes the appropriate stage, and packages it up into a message containing the name of the next stage, and then sends it to the registered name of the next process handing the appropriate stage.

- Note that although the loop/0 function is shared for each stage, independent instances of it exist for each.

```
% receive loop that hands work the the appropriate stage
loop() ->
  receive
    {double, N}   -> add ! {add, doubler(N)}, loop();
    {add, N}      -> divide ! {divide, adder(N)}, loop();
    {divide, N}   -> complete ! {complete, divider(N)}, loop();
    {complete, N} -> completer(N), loop()
  end.
```

Pipeline

```
1> c(pipeline).
{ok,pipeline}
2> pipeline:start().
true
3> registered().
[code_server,inet_db,erl_prim_loader,init,add,user,rex,
 error_logger,kernel_sup,global_name_server,kernel_safe_sup,
 file_server_2,global_group,complete,divide,double,
 application_controller]
4> double ! {double, 2}.
COMPLETE: 1.66667
{double,2}
```

Pipeline

We can see the names registered in the process registry.

```
1> c(pipeline).
{ok,pipeline}
2> pipeline:start().
true
3> registered().
[code_server,inet_db,erl_prim_loader,init,add,user,rex,
 error_logger,kernel_sup,global_name_server,kernel_safe_sup,
 file_server_2,global_group,complete,divide,double,
 application_controller]
4> double ! {double, 2}.
COMPLETE: 1.66667
{double,2}
```

Pipeline

```
1> c(pipeline).
{ok,pipeline}
2> pipeline:start().
true
3> registered().
[code_server,inet_db,erl_prim_loader,init,add,user,rex,
 error_logger,kernel_sup,global_name_server,kernel_safe_sup,
 file_server_2,global_group,complete,divide,double,
 application_controller]
4> double ! {double, 2}.
COMPLETE: 1.66667
{double,2}
```

We start the process by sending a value to the first stage (“double”) that contains the stage name and the argument. The result gets printed at the end.

Process dictionaries

- Sometimes you just need state for performance reasons. Not all algorithms work well in a purely functional model.
    - I would argue that this is the main reason for the ML family having a wider usage base in industry than Haskell.
- The process dictionary is a block of mutable state that each process contains. These are **not shared between processes**.
- The process dictionary is there if you absolutely and really need to program with side-effects.
    - You should always feel shameful and guilty if you use the process dictionary, and think hard about a way to implement the algorithm without it.

What haven't covered on Erlang?

- Error handling. What if a message comes in that doesn't match a clause in the receive block?

- How are errors propagated between processes? What if a process dies and another is waiting on it to send a message? Erlang was built to be robust to this and allow processes to gracefully deal with this situation.

- How does Erlang support hot-swappable code? This is pretty cool -- you can upgrade portions of a running program without bringing it down and restarting. Very useful for critical services (such as telecommunications systems).

- Standard library. Erlang has a nice standard library to do all kinds of useful things. We haven't touched on that at all.

Potential Synchronization Problem: Deadlock

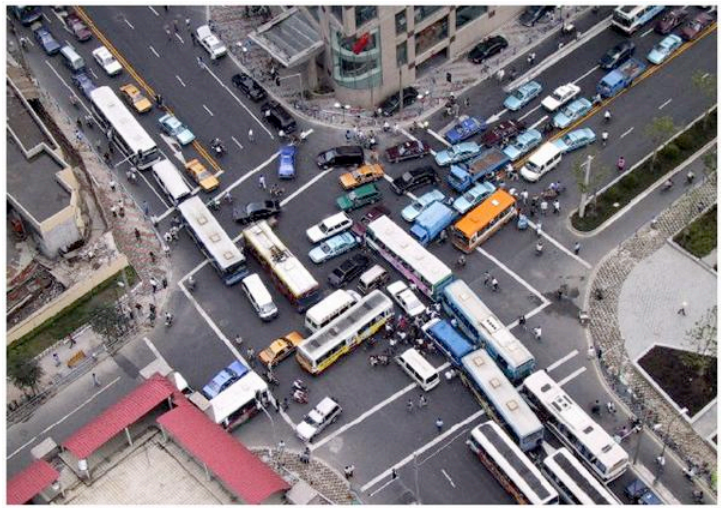
- Let’s add a function to the account example called **transfer()** that is based on withdrawing from one account and depositing into the other.

- Let’s design the algorithm such that the source of funds performs the transfer.

```
void transfer(Account src, Account dest,
              int i) {
  pthread_mutex_lock(src.lock);
  pthread_mutex_lock(dest.lock);
  int balance = src.getBalance();

  if (balance < i) {
    error(“Not enough money.”);
    pthread_mutex_unlock(src.lock);
    pthread_mutex_unlock(dest.lock);

    return;
  }

  balance = balance-i;
  src.setBalance(balance);

  balance = dest.getBalance();
  balance = balance+i;
  dest.setBalance(balance);
  pthread_mutex_unlock(src.lock);
  pthread_mutex_unlock(dest.lock);
}
```

Deadlock

- Now what if two threads attempt to transfer between two accounts, one from A to B and the other from B to A. What could possibly go wrong?
- A bad interleaving could result in:
  - **T=1: A** acquires a lock on itself.
  - **T=2: B** acquires a lock on itself.
  - **T=3: A** blocks because it cannot acquire a lock on **B** since **B** already locked itself.
  - **T=4: B** blocks for a similar reason.
  - We’re stuck.



Preventing deadlock

- Lock reordering
- “trylock” lock acquisition routines that attempt to acquire a lock, but if they fail immediately return instead of blocking. On failure, one can unlock potential sources of deadlock before retrying from the beginning.
- Thread analysis tools.
- Avoiding explicit locks.