Concurrent Programming in Education: Time for a Change

by

Christopher J Lieb

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

May 2011

APPROVED:

____

Professor Gary Pollice, Major Advisor

____

Professor Micha Hofri, Thesis Reader

____

Professor Craig Wills, Head of Department

Abstract

Writing concurrent programs using shared memory causes many programmers much trouble, due primarily to unsafe semantics. Memory corruption, race conditions, deadlocks, and even livelocks are trivially easy to introduce into a program and painful to hunt down due the nearly infinite possible interleavings of instructions between the threads.

Undergraduate curricula traditionally introduce students to the idea of shared memory multithreading in a systems programming or operating systems class, but rarely expose them to any alternate models of concurrent programming. This leaves them with the idea that shared memory is the only way to do concurrent programming.

After students were exposed to alternate models, they came to prefer them to the standard shared memory model. This happened despite their distaste of the programming language that was utilized in performing the study. The students also expressed interest in alternate models of concurrency being taught in the computer science curricula at WPI.

Acknowledgements

My advisor, Gary Pollice, for making sure that I read plenty of research papers and for making sure that I got everything done. Maybe someday I'll learn to write in the active voice naturally.

My reader, Micha Hofri, for volunteering to read my thesis and for giving me good and timely feedback.

Paul Kehrer and Peter Swire, for giving some of your time to test out the parts of my study that I would be giving to subjects. The mistakes that you guys found I'm sure saved the study's subjects untold amounts of grief and confusion.

Prof. Jeanine Skorinko of the Department of Social Science and Policy Studies, for letting me use some of her lab space for conducting the study, giving me useful advice about running studies, and for letting me advertise my study to students in her classes. Also, I'd like to thank/apologize to some of her research assistants and students, whom I had to kick out of the lab so I could run my study.

Prof. Joeseph Beck, for helping an inexperienced, amateur statician like myself to design a study and figure out ways to analyze the data to get the most useful results.

Last, but not least, my lovely fiancé, Sarah, for poking and prodding me to get my thesis done whenever my motivation was lagging. Thanks to you, I should actually graduate on time, despite my pessimistic view on the completion of my thesis.

Andreas Koeller, for designing the amazing LaTeX template that I used to produce this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computers originated with a sequential programming model: one instruction would execute, followed by the next instruction, followed by the next, until the end of the program is reached. While this model evolved with the introduction of decision structures, loops, and function invocations (jumps), it could still be conceptualized in the same sequential manner as far as execution models are concerned. Sequential execution served the computer industry well for decades, but was not always the optimal way to solve problems.

These problems involved tasks that could or should be run at the same time, such as vector operations or large-scale scientific simulations. In order to achieve these operations more efficiently, a parallel model of computing needed to be created. While this can be achieved on the hardware side by adding extra processing units to a computer (by adding extra sockets or by placing multiple units on the same piece of silicon), coming up with a good way to utilize multiple processors at once has proven to be challenging.

Time	Process 1	Process 2
1	a := obj.getProp()	
2	a = a + 1	
3		b := obj.getProp()
4		b = b + 2
5		obj.setProp(b)
6	obj.setProp(a)	

Figure 1.1: An example of concurrent access causing updates to be lost

[One outstanding problem in computer science] is concurrency: producing meaningful programming models for concurrency that are understandable by the masses of programmers as opposed to a few high priests of parallelism. ... Even the high priests at times get surprised by their own code today. (Anders Hejlsberg, creator of Delphi and C#/.NET Framework) [7, pg 311]

1.1 Concepts of Shared Memory Multithreading

While many models of concurrent programming have been proposed and implemented, the few that have taken hold in the computer industry leave much to be desired. The most prominent form of concurrent programming has become shared memory multithreading, an example of which is *pthreads* (the POSIX standard threading library). In shared memory multithreading, communication between threads is performed using memory that is shared between them. It is up to the programmer to figure out how to safely access memory from multiple threads without corrupting the state of the program stored in the shared memory. Figure 1.1 shows an example of memory corruption where an update made by process 2 is lost because process 1 overwrites it.

To work around this, the concept of locks was created. A lock is placed around a shared piece of memory, and it must be acquired in order to access that memory. In the case of mutexes (mutually exclusive lock), only one thread can have a given lock at a time; if another thread wishes to acquire a lock while it is being held, it must wait until the lock is

Time	**Process 1**	**Process 2**
1	getLock(a)	getLock(b)
2	getLock(b)	getLock(a)

Figure 1.2: An example of a deadlock

available again. Semaphores are similar to mutexes, except they allow a fixed number of threads to hold a lock simultaneously instead of only one. Once the fixed number of locks that the semaphore governs have been acquired, other threads must wait until one of those locks has been released.

Since locks are not a feature of the shared memory, they can be bypassed, either maliciously or accidentally, preventing them from doing their job. If a mutex is bypassed, multiple threads could have access to a piece of memory that is supposed to have only a single thread access it at a time. This would allow issues such as concurrent writing to the same memory location, causing the memory at that location to be corrupted.

Even if all accesses to the same piece of memory use a lock correctly, issues can still arise. A common issue is deadlock. This occurs when all threads are stuck waiting to acquire a lock. An example of this is shown in Figure 1.2, where two process try to acquire the same two resources, but in different orders. In this case, the processes will end up waiting on each other indefinitely as they each have a lock on a resource that the other needs. Situations like this can be prevented by having a thread release all locks it currently holds if it is unable to acquire all of the locks it needs.

A common thread with all of these techniques designed around shared memory multithreading is that they all require the programmer to write more code to make them work. Unfortunately, more code means more chance for error, and errors in multithreaded programs can lead to hangs over resource contention or strange behavior and crashes due to corrupted memory. To complicate matters further, these errors are some of the most difficult to track down due to the nearly infinite number of ways the instructions from the different threads can interleave. [13]

Certainly, dealing with parallel processes has produced the most difficult-to-deal-with bugs. (Guy Steele, contributor to Common Lisp, Scheme, Java, and creator of Fortress) [16, pg 362]

The multithreaded stuff, frankly, scares me... (Brendan Eich, creator of JavaScript) [16, pg 153]

... humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations. [18]

1.2 Prominence of Shared Memory Multithreading

Part of the reason that shared memory multithreading has become so prominent in industry has been the emphasis placed on it in undergraduate education. A common component of a computer science undergraduate curriculum is a course on operating systems, which typically covers concurrency issues through the dining philosophers problem and the producer-consumer (bounded buffer) problem, among other classic problems. (A book commonly used in operating systems courses, *Operating System Concepts* [17], covers bounded buffer, reader-writer, and dining philosophers.) In exploring solutions to these problems, a shared memory approach is taken, with no mention of any other technique, leaving students with the impression that there is no other way to deal with concurrent problems. For example, in *Operating System Concepts* [17], shared memory multithreading is the only presented technique for solving these concurrent problems.

This focus on shared memory multithreading leaves students with the impression that it is the only way to go about solving concurrent programming problems. The threading models that are built into Java and C#, two of the most used programming languages today, are implementations of shared memory multithreading. The pthreads library, the most common way to multithread C and C++ applications, is also based on shared memory multithreading. Given the industry's prominent use of shared memory multithreading, who would blame students for making this assumption?

Despite the industry's tendency toward shared memory multithreading, academics have conceived many different models of concurrency, some of which are starting to see exposure in increasingly mainstream languages. Erlang and Scala both support the actors model natively [3][9], and Go supports the channels model [2], an offshoot of communicating sequential proceses (CSP). Both of these models are based on the idea of message passing as the sole form of inter-thread communication, preventing access to any shared memory.

There are also other models, such as shared transactional memory (STM), which puts a transaction model around shared memory. This means that memory stays consistent for a single thread throughout a transaction, ensuring that there is no memory corruption. If a transaction fails, it can be backed out and attempted again, all without corrupting the program state. In this manner, it is much like transactional integrity that database management systems feature.

1.3 Importance of Concurrent Programming

Regardless of how we go about it, concurrent programming is an integral requirement for future students. This is due to the current trend in consumer processors, where clock speeds are staying relatively constant, while the number of processing cores on a single die is steadily increasing.

Up until the introduction of AMD's X2 processors in 2005, multi-processing unit personal computers were a rarity; multiprocessing was primarily the realm of large, special purpose computers used by the military and scientific community. Since the release of the X2 line, common consumer processors have sported multiple processing cores (between 2 and 8), with single core processors becoming more and more rare. At the same time, clock rates have stayed around 3GHz, ending the climb they had been on since the advent of Moore's law.

The end of clock rate escalation means that programs will no longer become faster just by throwing a better processor at the job since single core performance of processors is growing at a very slow rate. In order to take advantage of the current trend in processors, programs must be able to efficiently utilize multiple cores of the processor at once.

One area of research that is gaining prominence is the area of automatic parallelization of programs. This is either done as a library and pre-processor combination, like in OpenMP [1], which can be used with the C language, or as part of the language itself, as in Fortress [5]. These techniques tend to focus on parallelizing loops, though they have not shown the same performance benefits that one can achieve by parallelizing a program by hand [14].

All of this points to programmers needing to write concurrent programs in order to be able to fully utilize processors of today and tomorrow. The current undergraduate curriculum does little to prepare students for this reality, spending no more than a few weeks dealing with concurrency before moving on to another topic. At the same time, shared memory multithreading does little to make this task easier with the myriad of ways it allows programmers to make confounding mistakes.

This thesis will investigate the ability of students to learn and utilize models of concurrency in programs that are alternatives to the standard shared memory model. This investigation will center around the actors and channels models, both of which are based on the message-passing paradigm. The goal is to discover whether (a) students prefer an alternative model, (b) students are capable of learning an alternative model on their own, (c) students are capable of using an alternative model for designing and writing concurrent programs, and (d) students want to see alternative models included in their curricula.

Chapter 2

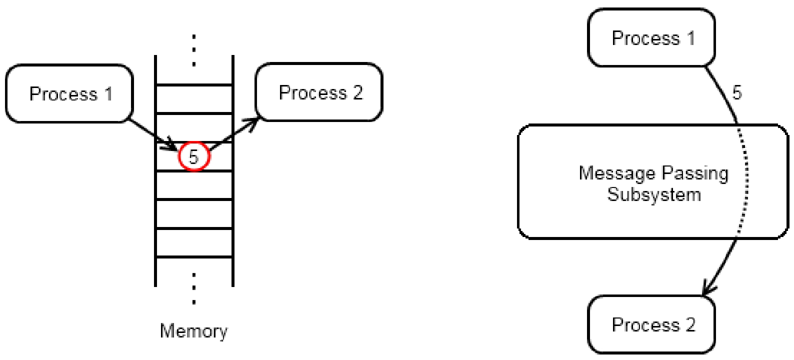
Background

When implementing concurrency in programs, communication is key. If threads do not need to communicate, then all models are equal in terms of safety and robustness. All that is needed are facilities to create threads and to end threads.

When inter-thread communication is added into the mix, different models of concurrency start to show their differences. As mentioned earlier, shared memory multithreading allows inter-thread communication by allowing threads to share the same piece of memory. (In order to communicate, one thread writes into a known memory location, then another thread reads from that same location, as seen in Figure 2.1a.) This sharing of memory is what leads to all of the common issues associated with shared memory multithreading: locks, memory corruption, etc.

One way to work around the issues of communication using shared memory is to perform communication using message passing, as seen in Figure 2.1b. This technique is much like interprocess communication (IPC) and message queuing systems (such as Java Message Service [6]). In its most common form, it is a library that runs on top of a shared memory multithreading system. This leaves the programmer the option to revert to shared memory-based communication whenever they like.

To make communication safer, different models of concurrency have grown around the



(a) Shared memory (b) Message passing

Figure 2.1: Different models of inter-thread communication

idea of message passing, completely removing the option of shared memory communication from being used. This forces the message passing facilities to have enough functionality to be able to handle all concurrent tasks. Two message passing models of concurrency that this paper will deal with are the channels (CSP) and actors models.

2.1 CSP and Channels

2.1.1 CSP

One alternate model of concurrent programming was created by C.A.R. Hoare and introduced in his 1978 paper *Communicating sequential processes* [12]. Hoare's primary idea was to make input and output basic primitives of the programming language. These primitives could be leveraged as the sole method by which processes could communicate. Since there is no shared state, many of the issues of shared memory multithreading are eliminated altogether. The technique that Hoare created shares the name of the paper that defined it: **Communicating Sequential Processes (CSP)**.

While Hoare's original idea treated the endpoints of a communication channel as language primitives and assigned unique names to the processes, he later reformulated the idea by making the communication channels themselves the primitives and making the processes anonymous [8]. Both formulations of CSP operated using rendezvous semantics, where a process could not send a message until the receiving process was ready to receive the message. This model was used as the basis or as part of some programming languages after Hoare's work was published (such as Newsqueak, Scala, and Go), though some implementations replaced the rendezvous semantics with asynchronous message passing. This reformulation is commonly referred to as channels.

2.1.2 Channels

These "channels" that were born from CSP are typed, one-way communication channels that can be used to allow one thread to send messages to another thread. A channel supports two operations: send and receive. The send operation on a channel is non-blocking, while a receive blocks until a message is available. A channel has an internal buffer of messages that are available to be received. This allows the sending thread to "send and forget", assuming there is still space available in the buffer. One language that implements channels as a primitive is Go, a new language from Google that was initially released in November 2009 and created by Robert Griesemer, Rob Pike (creator of Newsqueak), and Ken Thompson (creator of B, upon which C is based).

2.1.3 Go and Channels

Go is a compiled language that combines some of the syntax of C with some more dynamic aspects to try to form a next generation systems programming language. One interesting feature of Go is the built-in multithreading that it has. Go's multithreading is based on channels and *goprocess*es. *goprocess*es can be thought of as lightweight threads (threads that only exist within the language runtime that the operating system does not know about). A *goprocess* is created by prepending the keyword go to a function call; this creates a new thread that runs that particular invocation of the function. Go relies solely on its built-in

```
func main() {
  // create a channel that carries int's, can only buffer a
  // single item
  c := make(chan int, 1)
  d := make(chan bool, 1)

  // launch a new goprocess that runs the test method and
  // pass it the channels c and d
  go test(c, d)

  // enters the block that receives a message first
  // since c is sent a message by the goprocess, it's code
  // path is picked and the program prints the number 5
  select {
    case out1 := <- c:
      fmt.Printf(out1)
    case out2 := <- d:
      fmt.Printf(out2)
  }
}

func test(c chan int, d chan bool) {
  // put the int 5 into the channel
  c <- 5
}
```

Figure 2.2: A simple program in Go that uses channels

support for channels as the only way for *goprocess*es to communicate; no memory is shared between *goprocess*es from the programmer's point of view.

An example of *goprocess*es and channels can be seen in Figure 2.2. This example shows how *goprocess*es are spawned, how communicating through channels works, and how to do a multiplexed receive on a channel. This multiplexed receive makes Go's channels a very powerful way to communicate.

2.2 Actors

2.2.1 Actors

A second alternate model of concurrent programming came out of the artificial intelligence (AI) community in the 70's. The **actors model** was first proposed in 1973 as a way to easily parallelize AI tasks into a "swarm of bees" structure [11]. Each actor (bee) would perform a specific task and communicate using messages with other known actors. The original formulation of actors proposed that each actor be given its own processor on which to execute to allow actors programs to run more efficiently.

Actors are identified by a unique name and have only a single receive port that all other actors send messages to using the actor's name. It is the actor's job to know what to do with a message that it receives. This is usually accomplished using a form of pattern matching that is built into the programming language. Two languages that implement actors are Erlang (built into the language) and Scala (part of standard library).

2.2.2 Erlang and Scala

Erlang was one of the first prominent actors languages when it came out of Ericsson in 1993 [4]. It is a functional language, which extends to its native actors support. Figure 2.3 contains an example of Erlang code that performs a task similar to what is shown in Figure 2.2. This example also shows off the pattern matching, another native feature of Erlang, that makes it easy to correctly dispatch off of a received message.

One interesting feature of Erlang's actors model is its resilience to crashed actors. For each actor in Erlang, the runtime keeps track of all of the actors that it is linked to. If an actor crashes for some reason, a special abnormal exit message is sent to all actors linked to the crashed actor, allowing them to properly handle the crash.

Scala is a newer actors language that more closely follows the object oriented model of programming, though it has many functional programming features included. The design

```
-module(example)

main() ->
  %% create a new actor that runs test
  Test = spawn(example, test),
  %% send a message that includes myself so I can get a response
  Test ! (req, self()),
  %% receive one of the two messages, tagged by the labels c and d
  %% since c is received first, the number 5 is printed
  receive
    (c, I) ->
      io:format(" w n", [I]);
    (d, B) ->
      io:format(" w n", [B])
  end;

test() ->
  %% once a req message is received, send a c message with the
  %% number 5 attached
  receive
    (req, Target) ->
      Target ! (c, 5)
  end;
```

Figure 2.3: A simple program in Erlang that uses actors

of Scala's actors model is heavily influenced by Erlang, which leads to its actors features having a more functional flavor to them. The similarities can be seen in Figure 2.4, which is a fairly direct translation of the Erlang example in Figure 2.3. One thing to note is that Scala's pattern matching is not nearly as powerful as that of Erlang, forcing the programmer to create a different class for every type of complex message that they want to handle instead of being able to simply put them in a tuple.

2.2.3 Action for Go

Functionality

As was mentioned earlier, Go is a channels language; it has no support for actors out of the box. In order to add actors to Go, I developed a library called *action* that implements actors based on channels. The library tries to follow the style of Erlang actors when possible, but the fact that Go doesn't allow things like operator overriding prevents the library from having as nice a syntax as Erlang or Scala for actors functions.

action implements actors as methods that have a specific signature: they take an `Actor` struct and an array of elements of undefined type, and return nothing. The `Actor` object they are passed represents themselves and the array of elements is the array of arguments that were used to invoke the actor function. The array of untyped elements is necessary due to Go not having a way to explode an array into function arguments. To more easily deal with these untyped arrays of objects, a function called `InterpretArgs` is included, which allows you to pass references to declared variables and the original array and get the variables filled with the values out of the array after they have been type checked.

In order to send a message to an actor, you must call the `Send` function on an `ActorHandle`, which can be obtained as the result of a call to the `Spawn` function. In order to allow responses to messages, a message contains a handle for its sender. To allow this, calls to `Send` must include the `Actor` variable that is passed to an actor when it is created. The need for an `Actor` object in order to send creates a problem as the main method does not have an `Actor` object associated with it. To work around this, a `Run` method was created that is used to launch the first actor in a program's execution; all other actors are created using `Spawn`.

action implements two types of receive: standard and time-limited. The standard receive, `Receive`, blocks until a message matching one of the tags in the receive block is matched. The time-limited receive, `ReceiveTO`, acts identically to the standard receive, except it stops blocking after a specified amount of time and exits without any of the blocks in the receive statement executing.

An example of a Go program that uses *action* can be seen in Figure 2.5. This program is a translation of the Erlang program in Figure 2.3. You can see similarities in the structure of the code between the *action* implementation and both the Erlang and Scala (Figure 2.4) versions. Both *action* and Scala require a special, non-actor main method that simply launches a main actor. You can also see the way that message tags work as a primitive form of pattern matching.

Implementation

Behind the scenes, each actor has a single channel that it uses to receive messages. The `Send` function is a member of the `ActorHandle` type, which is used to make sure that the correct channel is used to send the message.

Messages are represented using the `Msg` type. The message has three components: the sender, the tag, and the body. The sender is simply an `ActorHandle` to the `Actor` that sent the message. The tag is a string used to identify the message; it is needed to work around the fact that Go does not have any pattern matching capabilities as part of the language. The body is an untyped object that is attached to the message; it is the job of the receiving actor to know what type of object is placed in the body and cast it as appropriate.

The code bodies of receives are implemented as maps, which is unfortunately exposed to users of the library. Using maps allows easy dispatch of received messages by using the tag of a message as a key into the receive map. It also makes the implementation of `ReceiveTO` relatively simple as all that is needed is to add a special element to the receive map and to launch a timeout actor that sends a message with that special tag that ends the receive statement without doing anything.

The *action* of receiving a message is implemented as a loop that periodically checks the actor's receive channel for messages. When one is found, the receive map is checked to see if the there is an entry for the given tag. If there is not one, the message is discarded, just like in other actors languages. If the tag is found, then the function that is mapped to it is executed and the receive exits. To prevent busyloops waiting on the correct message, a quick sleep is added to the loop so that the *goprocess* that the actor is running on can be descheduled so that another actor can run. Without this sleep, an actor can hog the processor waiting for the right message to arrive. Since it is hogging the processor and not allowing other actors to run, it will never receive the message it desires, causing the hogging to be indefinite.

The *action* library as a whole has proven to not run nearly as fast as the built-in channels, primarily due to the sleeping that is done as part of the receives. Fortunately, the study that required the construction of *action* does not require the library to be fast as student ability is being tested rather than execution speed of programs.

```
class Runner {
  def main(args: Array[String]): Unit = {
    val m = new Main()
    m.start()
  }
}

// create classes to represent each type of message
class Req(val sender: Actor) {}
class IntReply(val num: Int) {}
class BoolReply(val bool: Boolean) {}

// the actor that will initiate the request-response sequence
class Main extends Actor {
  def act() {
    // create the actor that we will communicate with
    val test = new Test()
    test.start()

    // send a message to the test actor
    test ! new Req(self)
    // receive one of the two types of messages
    // since the test actor sends an IntReply message, the first
    // case is followed
    react {
      case c : IntReply =>
        println(i.num)
      case d : BoolReply =>
        println(j.bool)
    }
  }
}

// the actor that Main will communicate with
class Test extends Actor {
  def act() {
    // when a Req is received, reply with an IntReply
    react {
      case req : Req =>
        req.sender ! new IntReply(5)
    }
  }
}
```

Figure 2.4: A simple program in Scala that uses actors

```go
func main() {
  action.Run(actorMain)
  action.FinishAll()
}

func actorMain(a *action.Actor, args []interface{}) {
  // create an actor that will send us a message
  t := action.Spawn(test)

  t.Send(a, "req", nil)

  //  wait  to  receive  a  message  either  tagged  "c"  or  "d"
  // since "c" arrives first, it is the one that is printed
  action.Receive(map[string]func(*action.Msg) {
    "c": func(msg *action.Msg) {
      fmt.Printf(msg.Body)
    },
    "d": func(msg *action.Msg) {
      fmt.Printf(msg.Body)
    }
  })

  action.Finish()
}

func test(a *action.Actor, args []interface{}) {
  action.Receive(map[string]func(*action.Msg) {
    "req": func(msg *action.Msg) {
      msg.Sender.Send(a, "c", 5)
    },
  })
  action.Finish()
}
```

Figure 2.5: A simple program in Go that uses actors

Chapter 3

The Study

In order to measure the ability of students to learn and utilize models of concurrency in programs that are alternatives to the standard shared memory model, a study was performed. This study involved undergraduate and graduate students, preferably ones that had previous experience with concurrent programming. At WPI, this would include any students that had completed Operating Systems (CS 3013), Distributed Computing Systems (CS 4513), or graduate-level Operating Systems (CS 502).

3.1 Design

The study was designed to test the ability of students to learn one of two message passing models of concurrency: actors and channels. The pool of participants was split in half, with each group being assigned one of the two models. The participants were given two weeks to learn the model of concurrency that their group would be using. To make study time as equal as possible, students were given exactly 14 days (with one exception) to study on their own.

At the start of the two weeks, each subject was given a link to the Effective Go tutorial. Since the tutorial teaches the channels model only, a special version of the document was created that substituted a section on actors written for this study in place of the section on channels that tried to go into the same amount of depth as the original section. Subjects in the actors group were also given an API reference *action*the library that was created using Go's `godoc` command. The goal was to have the *action* library look as official as possible so that the subjects in the actors group would not be tempted to brush it off due to its amateur appearance. Students were also provided with a shell account on a server that had both Go and *action* installed on it so that they wouldn't have to waste time trying to get Go setup on their personal machines or on shared machines they had access to.

At the end of the two weeks, each participant came in for a one-on-one meeting with myself where they demonstrated what they had learned by performing three tasks in succession: a baseline task, a design task, and a concurrent coding task. Each meeting lasted about an hour, and all tasks were videotaped for later review of subject performance. For each task, subjects were evaluated on time taken to complete, perceived ease of completion, and the quality of the answer produced. Each task also had a time limit, at which point the subject would be stopped and evaluated on the work completed up to that point.

While subjects were performing the task, they could ask the person running the study only questions to clarify the task; questions about Go were not allowed. Instead, the subjects were allowed free use of the Internet to try to find answers to issues they had. To prevent the wasting of time trying to figure out how to build their programs, subjects were given pre-made makefiles that handled the building and automated testing of programs that they wrote.

If, during the performance of a task, a subject started working on code that was unrelated to the task at hand, they were instructed to return to the task they were assigned unless they had a good reason for the deviation (such as writing their own test harness to test a smaller unit of code than the official test harness tested). This problem was encountered mainly with subjects writing code that was provided for them as part of a black box. Also, if a subject started trying to build the code by hand rather than to use the makefile, they were told of the presence of the make file.

3.1.1 Language Selection

In order to have results be comparable, a programming language that implemented both models was needed. While Scala implemented actors quite thoroughly, I found the implementation to be too complex as it had implemented actors as an object-oriented model. This created programs that were very verbose and needlessly complex in my opinion.

At the time the study was being designed, Google was just releasing its Go language. Go has native support for channels using a very simple, concise syntax. I found this syntax and model rather easy to learn. Unfortunately, Go did not have any actors implementation built-in or available from third parties. Luckily, with a working channels implementation, it is not difficult to build an actors library around it since the concurrency and communication are already implemented and tested, all that is needed is a different configuration. Given this, Go was chosen as the language to use in the study.

To remedy the lack of actors in Go, I wrote an actors library (patterned after the one in Erlang) that wrapped the channels model that was built into Go. Some workarounds had to be used to map the actors model from Erlang to a language with a static type system and without pattern matching, and the library was not very fast due to the way it handled receives. Despite the issues with the library, I did not feel that they would significantly impact the ability for students to learn the model.

3.1.2 Task Selection

In order to determine each subject's knowledge of their model of concurrency, tasks had to be developed. Since the language that was being tested was foreign to all of the subjects, a measure of just their knowledge of the language would be helpful in being able to differentiate between a subject who didn't understand the model versus a subject who might understand it but have issues using Go.

To measure a subject's ability with Go, a task needed to be developed that tested just their ability to use Go. It should not involve any concurrency, but instead consist of a simple, known programming problem. Sorting was chosen as it is a common problem seen in computer science classes and literature. Since there are many ways to perform sorting, it was decided that the subject should be told which to use as different sorts could take different amounts of time to implement and debug due to their differing levels of complexity. Further, to make sure that subjects were being tested only on their ability with Go and not their ability to remember sorting algorithms, a pseudocode implementation of the sorting algorithm would also be provided.

It was decided to test the subject's ability to use their concurrency model through two tasks: a design task and a coding task. The design task would present the subject with an algorithm that is easily parallelizable and ask them to create a concurrent design for it. The coding task would then ask them to implement all or part of a concurrent design. This would make sure that the student could both think in their concurrency model and use it in practice.

The Sieve of Eratosthenes was chosen as the algorithm that the subjects would work with. It is both easily parallelizable and fairly short to code. A bounded buffer (producer/consumer) problem was considered, but as it had no serial algorithm, was inappropriate for the design task.

3.1.3 Baseline Task

The purpose of the baseline task was to gauge the subject's familiarity and comfort with the Go language. This task was the same for both groups since no concurrency was involved. The subjects were allowed 15 minutes to perform this task.

The task had the subjects write a simple integer compare function (`Comp`)and a bubble sort function (`Sort`). The comparator would take two arbitrary integers and return 0 if they are equal, -1 if the first is less than the second, and 1 if the first is greater than the second. (This is very similar to Java's `compareTo` function.) The bubble sort was to do an in-place sort on a provided list of integers using a comparator that was passed to it; pseudo-code for

```
procedure bubbleSort( A : list of sortable items )
  n := length(A)-1
  for(a=0; a<= n; a++)
    for(b=n; b>a; b--)
      if A[b-1] > A[b] then
        swap (A[b-1], A[b])
      end if
    end for
  end for
end procedure
```

Figure 3.1: The bubble sort algorithm given to subjects

the algorithm (Figure 3.1) was provided to the subjects so that there would be no confusion as to what a bubble sort was or how one should be written.

In order for a subject to complete this task, they had to run their code using a provided test harness that would call the subject's `Sort` method with a known array of numbers and the subject's `Comp` function. The results of these calls were compared to hand-sorted arrays of the input numbers to ensure the subject's code was returning the correct values.

Expected Outcome

It is expected that the subjects should have little trouble with this task as it involves no concurrency and the algorithm is given to them; all they need to do is translate pseudocode to Go. It is expected that most subjects will take around 12 minutes to complete the task.

The answer provided by the subject was graded using the following criteria:

- `Comp` function
  - +1 correctly declared function
  - +1 correct return values
- `Sort` function
  - +1 correctly declared function
  - +1 correctly implement outer loop
  - +1 correctly implement inner loop

```
func Sort(lst []int, comp func(int, int) int) {
  n := len(lst) - 1
  for i := 0; i <= n; i++ {
    for j := n; j > i; j-- {
      if comp(lst[j-1], lst[j]) > 0 {
        lst[j-1], lst[j] = lst[j], lst[j-1]
      }
    }
  }
}

func Comp(obj1 int, obj2 int) int {
  if (obj1 < obj2) {
    return -1
  } else if (obj1 > obj2) {
    return 1
  }
  return 0
}
```

Figure 3.2: A solution to the baseline task

+1 correctly compare array elements

+1 correctly swap array elements

- +1 compiles

An answer like that seen in Figure 3.2 would receive full credit (8 points).

3.1.4 Design Task

The purpose of the design task was to gauge the subject's ability to use their model of concurrency for designing a concurrent solution to a problem. The only difference in the task between groups was which model of concurrency the group was told to use. Each subject was given 15 minutes to complete this task.

The task asked the subjects to give a high-level design for the Sieve of Eratosthenes, a prime sieve. They were provided with a description of how the algorithm worked (Figure 3.3) and asked to design a parallel version using their model of concurrency. The design

To find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:

1. Create a list of consecutive integers from two to n: (2, 3, 4, ..., n).
2. Initially, let p equal 2, the first prime number.
3. Strike from the list all multiples of p less than or equal to n. (2p, 3p, 4p, etc.)
4. Find the first number remaining on the list after p (this number is the next prime); replace p with this number.
5. Repeat steps 3 and 4 until p2 is greater than n.
6. All the remaining numbers in the list are prime.

Figure 3.3: The algorithm for the Sieve of Eratosthenes [19]

was to include how the task would be broken up into different threads and what information would be communicated between threads. They could express their design in any way they chose as long as it fit on a provided letter-sized sheet of paper.

As this task was to produce a design rather than an implementation, there was no way to test the solution produced by the subject. Therefore, it was entirely up to the subject to decide on when they had completed the task.

Expected Outcome

Since the students were tasked with designing a program, there was no truly objective way for the designs to be graded objectively. To try to work around this, I first translated a Newsqueak channels implementation of the Sieve from a Google Tech Talk given by Rob Pike [15] into Go. I then took that version and rewrote it using the *action* library for Go.

After these implementations had been created, I reverse engineered designs from them. For the channels version, a dataflow design was created (Figure 3.4). For the actors version, in order to capture the request/response nature of the program, a communication diagram was created (Figure 3.5).

Using these diagrams in conjunction with the solutions turned in by the subjects, a rubric

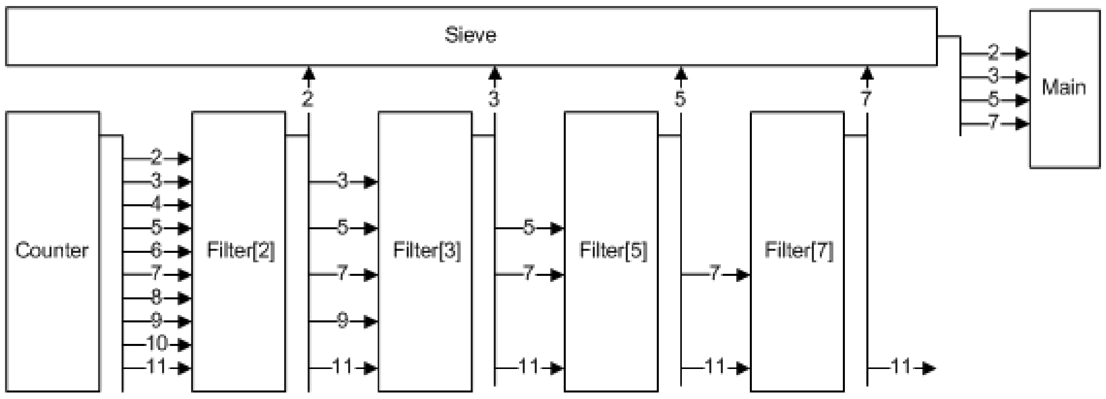


Figure 3.4: The channels dataflow design for the Sieve of Eratosthenes

was developed. The rubric was as follows:

5 Each filter is its own process, created as needed

4 Pre-created filters or filter for every number instead of every prime

3 All filters individual instead of chained (this is not how the algorithm works and creates synchronization issues)

2 Single filter process that filters by all discovered prime numbers

1 No concurrency used

0 No solution given

Each submission was matched to the description that most closely described it. In fact, most submissions were easily placed into one of these categories, helping to remove some of the subjectivity that I could have accidentally added to the process.

3.1.5 Concurrent Coding Task

The purpose of the coding task was to gauge the subject's ability to write code that used their model of concurrency. The only differences between the groups were the models used, the representation of the algorithm to code, and the starter code. Each subject was given 20 minutes to complete this task.

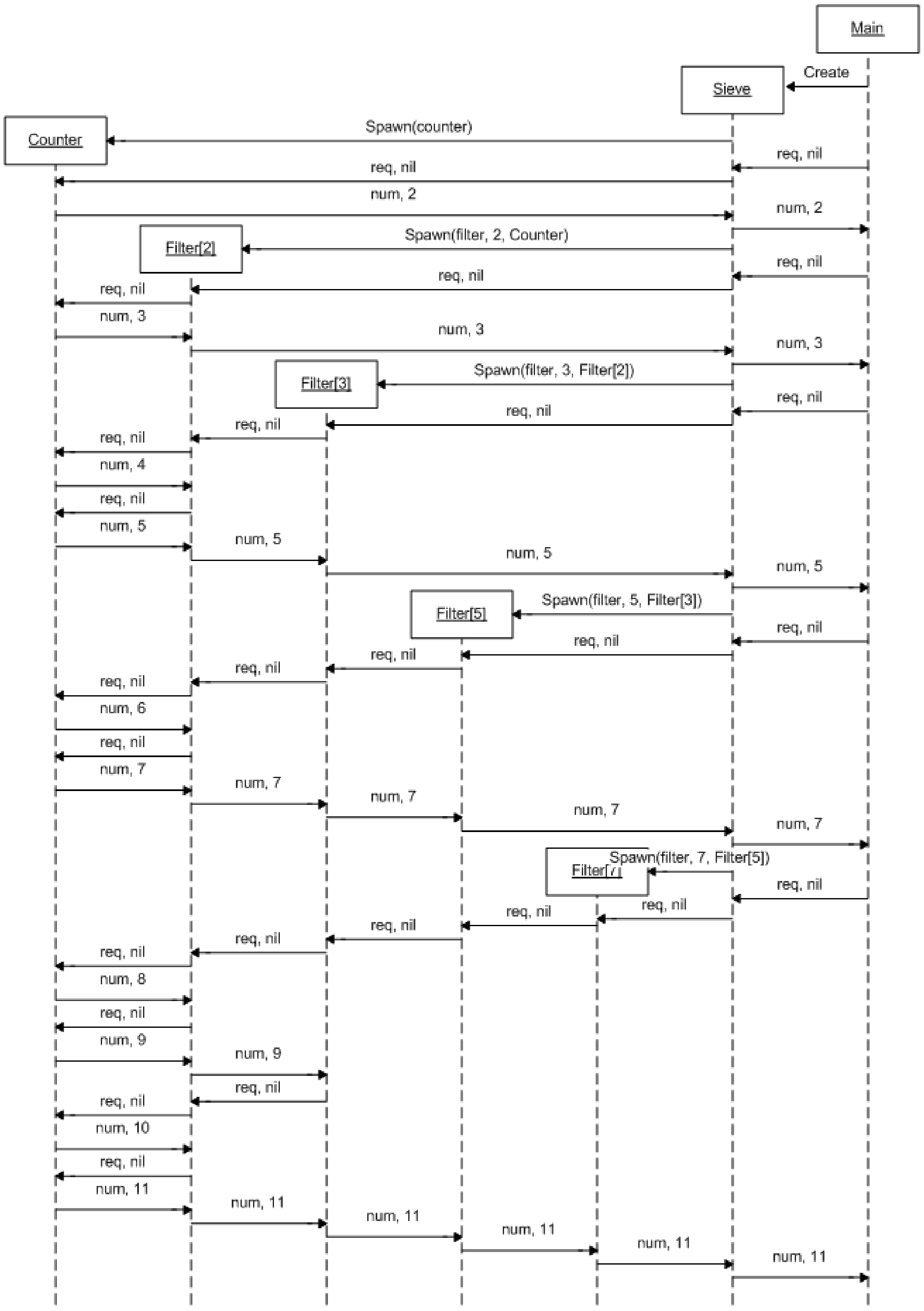


Figure 3.5: The actors communication diagram for the Sieve of Eratosthenes

The coding task required the subject to implement the `sieve` method of a solution to the Sieve of Eratosthenes that I wrote as part of developing the design task. Students in the channels group were given the dataflow diagram (Figure 3.4) and students in the actors group were given the communication diagram (Figure 3.5). The students were given the rest of the program from the implementation written by myself and told how the rest of the program worked via narrative description of the available API.

In order for subjects to complete this task, their program had to produce identical output to that specified in the task description. Since all of the output was handled by provided code, all the subject's code had to do was to produce the correct numbers.

Expected Outcome

It is expected that the subjects will have some trouble with this task as it requires them to not only write code that deals with concurrency, but it must interface with parts of a program that have already been written using a design that is likely different from the one that they just created. Also, if they were not comfortable using Go in the design task, then they were likely to be at a further disadvantage.

The answer provided by the subject was graded using the following criteria:

+1 create the counter process

+1 get a number from the counter/filter process

+1 send new prime

+1 create new filter process

+1 manage the counter and filter processes

+1 compiles

An answer like that seen in Figure 3.6 for channels and Figure 3.7 for actors would receive full credit (6 points).

```
func sieve(prime chan int) {
  c := make(chan int, 0)
  go counter(c)
  for {
    p := <-c
    prime <- p
    newc := make(chan int, 0)
    go filter(p, c, newc)
    c = newc
  }
}
```

Figure 3.6: A solution to the coding task for channels

3.2 Hypothesis

Coming into the study, I believed that the channels group would both out-perform the actors group and more strongly prefer their model. To test this, the null hypothesis that the two groups performed the same was used (difference of zero).

- **Hypothesis 1: Channels group will like their model more**

  Through experience preparing for the study using both models in multiple languages, I found that code using channels tended to lend itself to a fairly procedural style of coding, while actors tended to be more functional in style. As a teaching assistant, I witnessed many students balk at having to use functional programming languages such as Scheme in class, while having no or a favorable reaction to using procedural languages like Java or C. This led me to believe that students have a strong preference for languages that are procedural in nature. (While Java is an object oriented language, code within functions is still procedural.) Given this experience, I assume that students at WPI, in general, prefer programming languages that allow procedural programming and likely are more comfortable using them. This preference will likely cause the more procedural channels group to like their model more than the more functional actors group.

```
func sieve(a *action.Actor, args []interface{}) {
  last := action.Spawn(counter)
  for {
    var num int
    last.Send(a, "req", nil)
    a.Receive(map[string]func(*action.Msg){
      "num": func(msg *action.Msg) {
        num = msg.Body.(int)
        a.Receive(map[string]func(*action.Msg){
          "req": func(msg *action.Msg) {
            msg.Sender.Send(a, "num", num)
          },
        })
        last = action.Spawn(filter, num, last)
      },
    })
  }
}
```

Figure 3.7: A solution to the coding task for actors

- **Hypothesis 2: Channels group will perform better**

  Likewise, the students' comfort with programming in procedural styles, along with their perceived favoritism of procedural programming styles, will likely allow them to program more effectively using channels.

Chapter 4

Results

In total, 25 students signed up for the study. Of those 25, 17 completed the study, including having their meeting with me and filling out a post-study survey. Among the reasons given for the eight that did not complete the study were unexpected course loads, big interviews, and sickness. The remainder of this document will only concern the 17 subjects that completed the study.

The subjects in the study were split as evenly as possible between the two study groups: nine subjects were in the channels group and the remaining eight were in the actors group. The study was conducted in two runs, one after the other. No changes were made between runs other than the second group being given the suggestion to write a producer-consumer program as practice during their two weeks of learning. The first run consisted of nine subjects (five channels, four actors) and the second consisted of the remaining eight subjects (four channels, four actors).

Throughout this chapter, all references to t-tests or two sample t-tests are calculated using (4.1) with the null hypothesis, (H_0), that the two samples have a difference of zero. Any tests of statistical significance (P value) will use $\alpha = 0.1$ due to the small sample size. Any discussion of correlation of two values will use the scale defined in Table 4.1. All statistical analysis of data was done using Microsoft Excel 2010.

R **Value**	**Description**
$0.0 \leq \|R\| < 0.1$	No correlation
$0.1 \leq \|R\| < 0.3$	Weak correlation
$0.3 \leq \|R\| < 0.5$	Moderate correlation
$0.5 \leq \|R\| \leq 1.0$	Strong correlation

Table 4.1: Correlation value descriptions

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{\sigma_0^2}{n_0} + \frac{\sigma_1^2}{n_1}}} \tag{4.1}$$

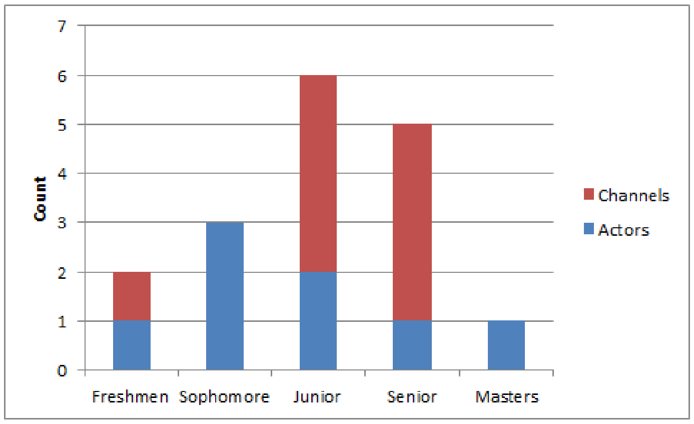
4.1 Demographics

The subjects were randomly assigned to study groups. The only time that the subjects were manually placed into a study group was when a subject had knowledge of one of the concurrency models but not the other. In this case, the subject was placed into the group whose model he or she was not familiar with. This was done three times, all because the subject had experience with actors.

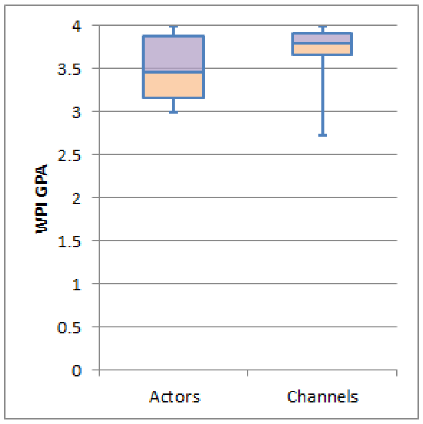
While it was preferred that subjects be a junior (3rd year) or higher, all class standings were allowed to participate. The actors group had a spread of class standings all the way from freshmen to masters students, with the largest number considering themselves sophomores and half of the group being below a standing of junior. The channels group was very highly concentrated at the junior and senior standings with only one subject being below the junior class standing. (Figure 4.1a)

Overall, the students that participated had strong GPAs at WPI. The actors group ranged from a 3.0 to a 4.0, with the average subject having around a 3.5 GPA ($\sigma \approx 0.42$). The channels group ranged from a 2.73 to a 4.0, with the average subject having around a 3.7 GPA ($\sigma \approx 0.40$) (Figure 4.1b). A two sample t-test is unable to reject the null hypothesis that the groups have the same average GPA ($P \approx 0.29$).

The prior experience of subject varied greatly in a variety of categories. As seen in



(a) Number of Students per Class Standing



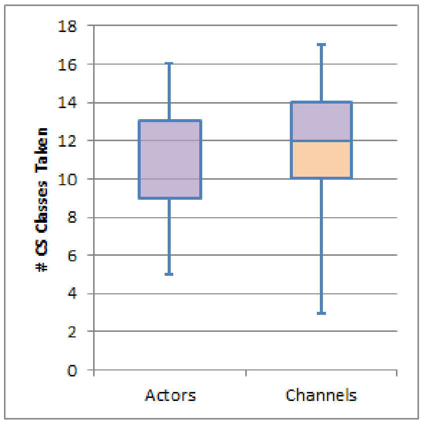
(b) WPI GPA (4 point scale) per Group

Figure 4.1: Study population demographics - Academics

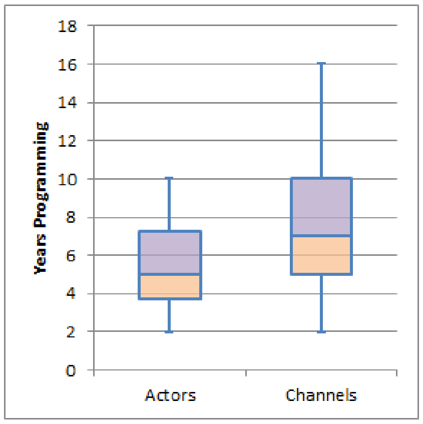
Figure 4.2a, subjects had taken between three and 17 computer science (CS) classes in their lifetime. (The numbers were not restricted to those taken in college or at WPI, so they could include classes taken in high school as well.) We were unable to reject the null hypothesis that the two groups had taken the same average number of CS classes using a two sample t-test ($P \approx 0.33$).

There was a similar spread when measuring the years of programming experience, which ranged from two years to 16 years (Figure 4.2b). No guidance was given as to what counts as a year of programming experience. Using a two sample t-test, we were unable to reject the null hypothesis that the groups had the same average amount of programming experience ($P \approx 0.19$).

Among the participants, C and Java were the most common programming language given when the subjects were asked to list all of the programming languages that they know, with all 17 subjects listing both. Following closely with 14 mentions is C++, though that number as well as the number for C are slightly questionable as four subjects gave C/C++ as an answer, which was counted as a response for each language. Due to the introductory computer science course at WPI (CS 1101) using Scheme as its language, 12 of the subjects had experience with the functional language. This meant that 100% of the study population had experience with imperative languages and about 76% had experience with functional



(a) Number of CS Classes Taken per Group



(b) Number of Years Programming per Group

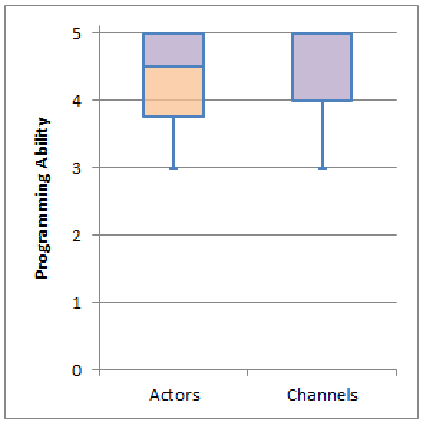
Figure 4.2: Study population demographics - Experience

languages. (In addition to the 12 subjects who cited experience with Scheme, one additional subject cited experience with both F# and OCaml.)

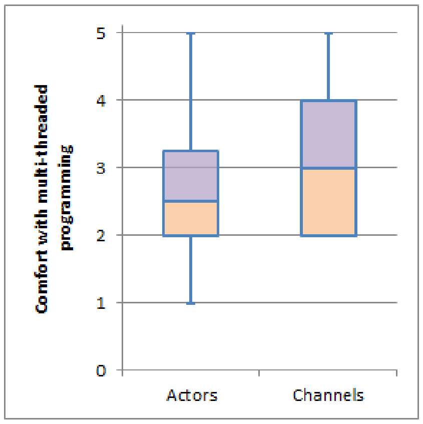
The subjects were asked to rate their ability to program, as well as their comfort levels with multi-threaded programming, C programming, Java programming, pthreads (with C) programming, and Java Threads programming. Overall, the subjects were very confident in their abilities (Figure 4.3a), with the population averaging a 4.24 out of 5 ($\sigma \approx 0.83$). The groups rated themselves virtually identically, as a two sample t-test failed, by a wide margin, to reject the null hypothesis that the groups were the same ($P \approx 0.40$).

Given the strong belief that the subjects had in their abilities, they didn't rate their comfort with multi-threaded programming very high (Figure 4.3b). The population had a mean of 3 out of 5 ($\sigma \approx 1.17$), with a difference of about half a point between the groups' averages. (Channels was the higher of the two.) Again, the null hypothesis of the groups being the same was not rejected ($P \approx 0.29$).

The results of the comfort levels with C and Java were strikingly similar, with both having a mean of 4.05. (The standard deviations were slightly different, at $\sigma \approx 0.91$ and $\sigma \approx 0.75$, respectively.) This similarity carried over to the comfort levels with pthreads and Java Threads, with both having a mean of 2.59 ($\sigma \approx 1.28$ for pthreads, $\sigma \approx 1.23$ for Java



(a) Programming Ability per Group



(b) Comfort with Multi-threaded Programming per Group

Figure 4.3: Study population demographics - Self-assesment

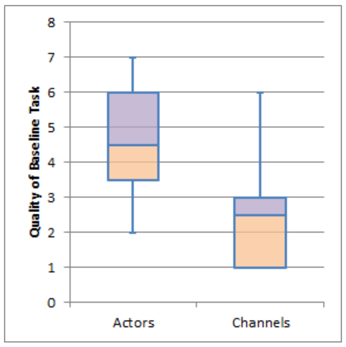
Threads).

4.2 Performance

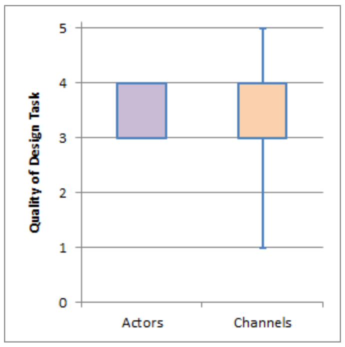
Before the first run of the study, it was planned that subjects would be measured by the time it takes them to complete each task and the ease with which I perceived them to accomplish the task. After the first run of the study, it was apparent that the time taken to complete a task would be a poor measure of performance. This is due to the fact that eight of the nine subjects in the first run were unable to complete the baseline task in the allotted time. Even worse, none of the subjects completed the concurrent coding task in the allotted time. The second run was not much better, with only two of eight completing the baseline task and one of eight completing the concurrent coding task.

To be able to get useable data out of these tasks, I decided to grade the code/design produced by each student, using the rubrics presented in Chapter 3. The metric of perceived ease of completion was retained as it was still useful even with the subjects' inability to complete the baseline and coding tasks.

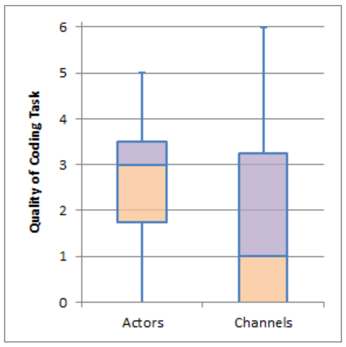
Through analysis of the scores and perceived ease measurements for all three tasks



(a) Baseline Task Score per Group (max 8)



(b) Design Task Score per Group (max 5)



(c) Coding Task Score per Group (max 6)

Figure 4.4: Individual Task Scores

over the whole study population, it was observed that the ease of completion scores closely tracked with the score assigned for each task. Because of this, only analysis of the score will be discussed in this section.

4.2.1 Baseline Task

For the baseline task (Figure 4.4a), out of a possible 8 points, the channels group had an average score of 2.5 ($\sigma \approx 1.69$) and the actors group had an average score of 4.5 ($\sigma \approx 1.85$). A two sample t-test was able to reject the null hypothesis that the groups had an equal performance ($P \approx 0.03$).

When subjects were asked to rate their comfort with Go, their answer was moderately correlated with their performance on this task ($R \approx 0.45$), with both groups having similar R values. Subjects' comfort with Java had weak negative correlation over the entire population ($R \approx -0.26$) and no correlation ($R = 0$) for this channels group. For the actors group, however, there was a strong negative correlation between comfort with Java and performance on the baseline task ($R \approx -0.51$).

4.2.2 Design Task

For the design task (Figure 4.4b), out of a possible 5 points, the channels group had an average score of 3.375 ($\sigma \approx 0.52$) and the actors group had an average score of 3.56 ($\sigma \approx 1.24$). A two sample t-test was unable to reject the null hypothesis that the groups had an equal performance ($P \approx 0.22$).

Subjects' ratings of their comfort with Go with their performance on the design task were weakly correlated ($R \approx 0.24$). When examining just the channels group, a strong positive correlation between the two was found ($R \approx 0.57$). Conversely, the actors group had a weak negative correlation ($R = -0.2$).

Ratings of comfort with Java were even more opposite than those with Go. While the overall population had virtually no correlation between comfort with Java and performance on the design task ($R \approx 0.05$), the channels group had a moderate positive correlation ($R \approx 0.43$) while the actors group had a strong negative correlation ($R \approx -0.79$).

When subjects were asked to rate their comfort with their assigned model of concurrency, a moderate positive correlation was observed ($R \approx 0.43$). This correlation was echoed by both the actors ($R \approx 0.38$) and channels ($R \approx 0.52$) groups.

It was observed that subjects' comfort with shared memory multithreaded programming models (pthreads and Java Threads) had little bearing on their ability to design concurrent programs using actors or channels. The whole population had a weak positive correlation between comfort with pthreads and performance on the design task ($R \approx 0.17$), which was echoed in the correlations of the actors ($R \approx -0.09$) and channels ($R \approx 0.23$) groups. Comfort with Java Threads showed itself to be a bad thing, with the whole population showing a weak negative correlation ($R \approx -0.25$) and the actors group showing a moderately week correlation ($R \approx -0.45$).

When comparing a subject's performance on the baseline task to their performance on the design task, a slight positive correlation was observed for the population ($R \approx 0.28$). While the actors group had virtually no correlation between the two statistics ($R \approx -0.07$), the channels group had a strong positive correlation ($R \approx 0.68$).

4.2.3 Concurrent Coding Task

For the coding task (Figure 4.4c), out of a possible 6 points, the channels group had an average score of 1.875 ($\sigma \approx 2.23$) and the actors group had an average score of 2.75 ($\sigma \approx 1.75$). A two sample t-test was unable to reject the null hypothesis that the groups had an equal performance ($P \approx 0.17$).

Subjects' ratings of their comfort with Go with their performance on the design task were weakly correlated ($R \approx 0.20$). When examining just the channels group, a moderate positive correlation between the two was found ($R \approx 0.34$). Conversely, the actors group had a almost no correlation ($R = 0.04$).

Subjects' comfort with C, while showing almost no correlation for the full population ($R \approx -0.11$), caused diverging correlations between the two study groups. The channels group showed a moderate positive correlation ($R \approx 0.42$) while the actors group showed a strong negative correlation ($R \approx -0.61$). Oddly, subject's comfort with Java showed a reversed pattern: the channels group had weak negative correlation between comfort with Java and score on the coding task ($R \approx -0.17$), while the actors group had a very strong correlation ($R \approx 0.81$).

Unexpectedly, subjects' comfort with their concurrency model did not always positively correlate with their score on the coding task. The population as a whole has a weak positive correlation ($R \approx 0.18$) and the channels group had a moderate correlation ($R \approx 0.45$). Oddly, the actors group had a weak negative correlation ($R \approx -0.27$).

Unlike with the design task, comfort with traditional concurrent programming models had a stronger correlation with performance on the coding task. While the population as a whole showed very weak correlation between comfort with pthreads and their score on the coding task ($R \approx 0.04$), the actors group showed a moderate negative correlation ($R \approx -0.42$) while the channels group showed a moderate positive correlation ($R \approx 0.47$). The correlations in regards to comfort with Java Threads were reversed: the channels group showed a weak negative correlation ($R \approx -0.13$) while the actors group showed a strong positive correlation ($R \approx 0.62$).

When comparing a subject's performance on the baseline task to their performance on the coding task, a strong positive correlation was observed for the population ($R \approx 0.58$). Both groups had positive correlations, but the channels group had a much stronger positive correlation ($R \approx 0.78$) than did the actors group ($R \approx 0.31$).

4.2.4 Total Score

In order to summarize a subject's performance over all tasks, a metric was created which used the results of all of the tasks to create a single value. Since the evaluation score was the most objective and useful of the three metrics collected for each task (time for completion, observed ease of completion, evaluation score), the values of it were aggregated into a single statistic, the total score. It was decided to weight all tasks equally as no emphasis was placed on one task over the others during testing. Given this, the metric of total score was calculated using (4.2).

$$score_{total} = \left[\left(\frac{score_{baseline}}{8}\right)\frac{1}{3} + \left(\frac{score_{design}}{5}\right)\frac{1}{3} + \left(\frac{score_{coding}}{6}\right)\frac{1}{3}\right] * 100\% \quad (4.2)$$

Using the total score metric, out of a possible score of 100, the actors group averaged 56.5 ($\sigma \approx 12.4$) and the channels group averaged 45.1 ($\sigma \approx 24.9$). A two sample t-test was unable to reject the null hypothesis that the two groups had the same total score ($P \approx 0.19$). (Figure 4.5)

Comfort with Go had a positive effect for both groups, showing a slight positive correlation for the actors group ($R \approx 0.26$) and a moderate positive correlation for the channels group ($R \approx 0.49$). Comfort with C had a mixed effect, having almost no correlation for the

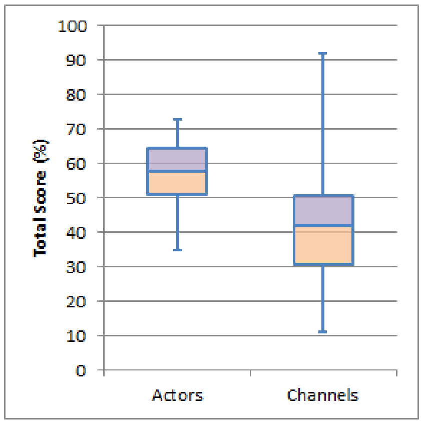


Figure 4.5: Total Score per Group

whole population ($R \approx -0.07$), but having a strong negative correlation for the actors group ($R \approx -0.72$) and a moderate positive correlation for the channels group ($R \approx 0.43$). Comfort with Java had a similar but opposite effect, with a moderate positive correlation for the actors group ($R \approx 0.49$) but virtually no correlation for the channels group ($R \approx -0.03$).

Comfort with channels or actors produced a positive correlation for all groups in the study (channels: $R \approx 0.53$, actors: $R \approx 0.21$). Similar to the pattern seen with comfort with C, comfort with pthreads had no correlation for the whole population ($R \approx -0.01$), while the actors group had a strong negative correlation ($R \approx -0.72$) and the channels group had a moderate positive correlation ($R \approx 0.43$). Comfort with Java Threads had the opposite effect, with a weak positive correlation for the actors group ($R \approx 0.24$) and a moderate negative correlation for the channels group ($R \approx -0.33$).

The subject's GPA had a weak positive correlation with their total score for the entire population ($R \approx 0.25$), with moderately positive correlations for both actors ($R \approx 0.37$) and channels ($R \approx 0.35$) groups.

The number of years that the subject had been programming had an even stronger correlation with the total score, with the actors group having a strong positive correlation ($R \approx 0.65$) and the channels group having a moderate positive correlation ($R \approx 0.49$).

Interestingly, the number of computer science courses that the subjects reported having taken adversely effected their scores, with a strong negative correlation for the whole

Year	Value
Freshmen	1
Sophomore	2
Junior	3
Senior	4
Masters	5

Table 4.2: Mapping of class standings to values used to determine correlation

population ($R \approx -0.60$) and an even stronger negative correlation for the channels group ($R \approx -0.73$). This trend was echoed (somewhat expectedly) in the correlation between total score and the subject's class standing at WPI. In order to find the correlation involving class standing, the standings were mapped to integers using Table 4.2. Using this mapping, a moderately negative correlation was found for the whole population ($R \approx -0.41$) and a strong negative correlation was found for the channels group ($R \approx -0.66$).

These odd results were caused by a single subject, a freshman, who had the highest total score of the entire population by a large margin. This subject had a score of 92%, while the rest of the scores were between 11% and 73%. Removing the data point created by this student changes the negative correlations observed previously with correlations that range from very weakly positive to moderately positive, which makes much more sense.

One thing that was unanimous was the strong positive correlation between subjects' ratings of their programming ability and their total scores. For the whole population, a strong positive correlation was observed ($R \approx 0.61$). Both the actors group ($R \approx 0.67$) and the channels group ($R \approx 0.65$) showed the same strong correlation.

Another positive was the correlation between the subject's self-reported time spent learning Go and their concurrency model and their total score. The whole population showed a moderate positive correlation ($R \approx 0.46$), the actors group showed a strong positive correlation ($R \approx 0.70$), and the channels group showed a moderate positive correlation ($R \approx 0.35$).

13 of the 17 subjects in the study received extra credit in at least one class for participating in the study. The average score of subjects not receiving extra credit was 76 ($\sigma \approx 14.7$),

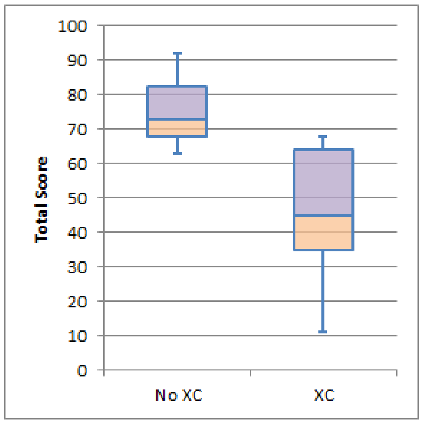


Figure 4.6: Total Score by Receipt of Extra Credit

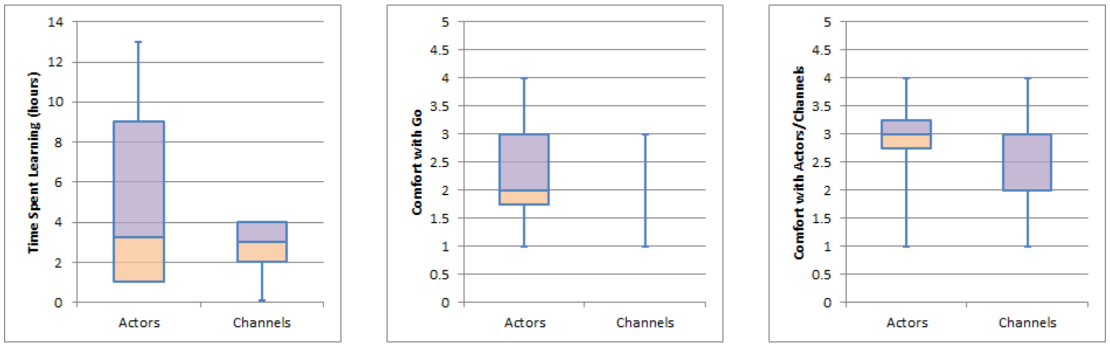
while the average score of subjects that did receive extra credit was 45 ($\sigma \approx 16.3$). A two sample t-test was able to reject the null hypothesis that the two groups had the same average ($P \approx .35e - 4$). (Figure 4.6)

4.3 Feedback

After a subject's meeting with me, they were given a post-study survey to fill out. On this survey, subjects reported that they spent an average of 3.89 hours ($\sigma \approx 3.74$ hours) learning the Go language and their model of concurrency over the two week period. The actors group spent, on average, 5.31 hours ($\sigma \approx 5.01$) preparing while the channels groups spent 2.62 hours ($\sigma \approx 1.48$) hours preparing (Figure 4.7a). Using a two sample t-test, we are unable to reject the null hypothesis that the two groups spent the same amount of time preparing ($P \approx 0.14$).

Subjects also reported how comfortable with Go they were, on a scale of one to five, at the end of the study. The actors group had an average of 2.25 ($\sigma \approx 1.04$), while the channels group had an average of 2 ($\sigma \approx 0.71$) (Figure 4.7b). A two sample t-test was unable to reject the null hypothesis that the two groups had the same comfort level with Go ($P \approx 0.34$).

The survey also asked subjects to rate their comfort with their model of concurrency on



(a) Time Spent Learning per Group (b) Comfort with Go per Group (c) Comfort with Actors/Channels per Group

Figure 4.7: Post-study survey results

a scale of one to five. The actors group rated their comfort with the actors model with an average of 2.88 ($\sigma \approx 0.99$), while the channels group rated their comfort with the channels model with an average rating of 2.56 ($\sigma \approx 1.01$) (Figure 4.7c). A two sample t-test was unable to reject the null hypothesis that the two groups had equal comfort with their respective models ($P \approx 0.16$).

The survey also included a series of yes/no questions relating to the study. A summary of the responses to these questions is provided in Table 4.3. The percentages are the percent of the population of the given group that answered "Yes" to the question.

Question	**Actors (%)**	**Channels (%)**
Is Go easier than C?	50.0	0.0
Is channels/actors easier than pthreads?	75.0	44.4
Is channels/actors easier to debug than pthreads?	37.5	44.4
Is Go easier than Java?	37.5	11.1
Is channels/actors easier than Java Threads?	62.5	44.4
Is channels/actors easier to debug than Java Threads?	75.0	44.4
Should channels/actors be included in the CS curriculum?	75.0	100.0

Table 4.3: Subject survey response summary

Subjects who answered "Yes" to including channels or actors in the computer science curriculum were asked how the topic would be added the curriculum: as a new class or as an addition to an existing class. Of the 15 subjects that thought that actors or channels should be added to the curriculum, eight thought that the content should be added as part of a new course and seven thought that it should be added to an existing course. Two subjects felt that the content should be added to the distributed systems course (CS 4513), two felt that it should be added to the operating systems course (CS 3013), and two felt that it should be added to the systems programming course (CS 2303). (Subjects were allowed to list multiple classes.)

Subjects were also given a space where they could give any additional comments they had about the study. Here are some responses:

> *I thought that the idea behind channels and actors was very intuitive. However, the syntax of the language and the way it handles types is quite unintuitive. The design of the language could have been much better.*

> *I really liked how easy it was to spawn new threads in Go, and also really liked Channels. However, I did not like go as a language.*

> *I am extremely comfortable with C, and thread-based systems are very natural to me. That said, channels to [sic] intrigue me...*

> *Pthreads aren't fun...this was a great new way to do concurrency*

> *Even if my skills in go were not good I think the overall idea of concurrency models is easier than pthreads.*

> *The idea of being able to run a process in the background (such as using & in unix) and the idea of flow between processes makes a lot more sense to me than pthreads and java threads. It seems like you focus more on the communication between actors in an operation, rather than how to initialize and maintain the individual actors.*

Overall, the comments from the subjects were positive, as seen above. The remaining comments consisted of subjects apologizing for not having spent more time preparing and acknowledging that they did not do as well as they feel they could have.

One subject was overwhelmingly negative in his comments about both Go and the actors model. His issues seemed to stem from having issues understanding how the actors model and the *action* library worked. He also had issues due to a misunderstanding as to how *action* worked concerning spawning new actors, which made the library seem much more limited. (More specifically, he thought that only the main actor could spawn new actors.) He also took issue with the choice of the Sieve or Eratosthenes as the choice of concurrent problem for the study. He felt that the design that was he was given was sequential in nature, even if it used multiple processes. In a way, this was true for the actors group since a request/response model was used for communication.

Chapter 5

Conclusions and Recommendations

This thesis set out to see if alternate forms of concurrency, primarily message-passing models such as channels and actors, would be easier for students to learn and make use of. This was to be measured using four criteria:

a **Students prefer an alternative model** From the post-study survey, subjects reported that 58% of them found their model to be easier than pthreads and 53% found their model easier than Java Threads. Further, the feedback from students that participated in the study was mostly positive, with six of ten responses being favorable of channels or actors. (Of the remaining three, one was negative and two did not mention whether they liked their model or not.) Two of those responses indicated that the subject liked their model of concurrency even though they did not like the Go language.

b **Students are capable of learning an alternative model on their own**

c **Students are capable of using an alternative model for designing and writing concurrent programs** These two goals dovetailed with each other, as a subject needed to learn the model on their own in order to be able to design and write concurrent programs using it as no interactive instruction was given. While the performance of the subjects was not as good as expected, they all demonstrated that they were able to learn some about their assigned model, with subjects averaging a score of 51% on all of their tasks. If they had learned nothing, their scores would have been much closer to 0%. (A subject that admitted to spending only 15 minutes preparing scored 11%.)

d **Students want to see alternative models included in their curricula** Subjects indicated through the post-study survey that 88% of them favored adding alternate models of concurrency to the curriculum. Of these, about half favored the creation of a new course about concurrency while the other half favored adding topics of alternate models of concurrency to existing classes, such as systems programming, operating systems, and distributed systems.

Given the results of this study, the increasing prevalence of multicore processors, and the decreasing rate of speed increases in processors, I feel that alternative models of concurrency should be added to the computer science curriculum. It is important that computer science education help prepare students for the world of computing, and that world is becoming increasingly parallel.

Technology doesn't scale, Computer Science scales. [10]

It is time that computer science education scaled to meet the needs of modern computing. I feel that we should start this by creating an experimental 4000-level course at WPI in concurrent programming techniques. This would follow the trend of WPI adding 4000-level computer science classes in areas that are becoming increasingly important, such as web programming (CS 4241). Adding this class would help better prepare WPI graduates to tackle problems in a more efficient concurrent manner, and to be able to produce working solutions to these problems in a faster and more robust manner.

5.1 Future Work

In order to present a stronger case for creating a new class in the computer science program, the experiment could be repeated, both at WPI and at other universities. This would help increase the population that we are gathering data from as well as dilute any effects that WPI might have on students that affect the results. It is possible that running this study at a school with a lesser computer science program might produce a population that is less interested in exploring more advanced topics in computer science.

If interest in alternate models of concurrent programming is still strong after widening the population of the study, then a class in models of concurrent programming should be developed. The class would be taught as a survey course, in the style of the programming languages course at WPI (CS 4536).

This class could start with shared memory multithreading, and then focus on two topic areas: automatic parallelization and explicit parallelization. The section on automatic parallelization could use OpenMP or Fortress as examples of languages that try to automatically parallelize loops in your code. It could look at the issues apparent in this style, such as data dependency preventing parallelization from occurring. From there, the section on explicit parallelization could cover message passing models and STM, to name a few. Students could see how performance can be greater when explicitly parallelizing code versus having the compiler parallelize loops for you.

Based on the success of the course, measured both by student performance and by student feedback over the course of the term, it could be made into a normal, non-experimental course which gives students an introduction to many types of concurrent programming. Since concurrent programming in computer science is starting to pick up the pace of innovation, the class will need to keep up with the current state of the industry. This means that the course will need to be updated frequently to keep it up to pace.

Appendix A

Task Descriptions

A.1 Baseline Task

Time: 15 minutes

Write a simple program to sort items in n array. Start with the file bench_sort.go.

The program must contain two methods:

func Comp(?, ?) ?

This function will take two integers and compare them. It will return (an int) -1 if obj1 is less than obj2, 0 if they are equal, and 1 if obj1 is greater than obj2

func Sort(?, ?)

This function will take two parameters: first, the array of integers to sort, and second, the comparator to sort them by. The comparator should use the same signature as Comp. This function will sort the items in the list from least to greatest using the bubble sort algorithm below. Sorting must use the comparator that is passed as the second parameter. Here is the pseudocode for the bubble sort algorithm if you are unfamiliar with it:

```
procedure bubbleSort( A : list of sortable items )
  n := length(A)-1
  for(a=0; a<= n; a++)
    for(b=n; b>a; b--)
      if A[b-1] > A[b] then
        swap (A[b-1], A[b])
      end if
    end for
  end for
end procedure
```

Testing

For your code to be finished, it must pass a test suite that has been provided. To run the test suite, run make test. If it outputs the text PASS, you are done. If it outputs FAIL, you need to fix your code.

A.2 Design Task

Time: 15 minutes

Design a prime number sieve using [actors/channels]. Each prime number should be used to construct a filter that only allows numbers through that it does not evenly divide, and should be its own [actor/process]. In other words, if you have found N primes so far, there should be at least N [actors/processes] currently running. You should create [an actor/a process] for each task that is being performed in the program.

Please include a description of the function of each [actor/process] that you would use and a brief sketch of how the [actors/processes] interact with each other. Your performance will be measured by the sensibility of your design. Sequential (serial) designs are not permitted.

Background

The Sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to a specified integer. Here is a description of a sequential (serial) version of the sieve:

> *To find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:*
>
> 1. *Create a list of consecutive integers from two to n: (2, 3, 4, ..., n).*
> 2. *Initially, let p equal 2, the first prime number.*
> 3. *Strike from the list all multiples of p less than or equal to n. (2p, 3p, 4p, etc.)*
> 4. *Find the first number remaining on the list after p (this number is the next prime); replace p with this number.*
> 5. *Repeat steps 3 and 4 until p2 is greater than n.*
> 6. *All the remaining numbers in the list are prime.*

Description of Sieve of Eratosthenes is taken from Wikipedia:
https://secure.wikimedia.org/wikipedia/en/wiki/Sieve_of_Eratosthenes

A.3 Coding Task - Channels Group

Time: 20 minutes

Write the **`sieve`** method for the channels implementation of the Sieve of Eratosthenes. Write your code in the file **`prime_sieve.go`** and compile it by running **`make`**. The file **`prime.go`** contains the remainder of the program and is required to build the program, but you are not allowed to look at its contents.

Expected Results

```
user@TIKI1 ~/coding $ ./prime
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191
193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283
293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401
409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509
521 523 541
```

Details

The sieve dedicates a process to each filter, one process for the counter (number generator), and one process for the sieve. The sieve is the creator of the other processes. All processes push their results into a synchronous (0-size buffer) channel.

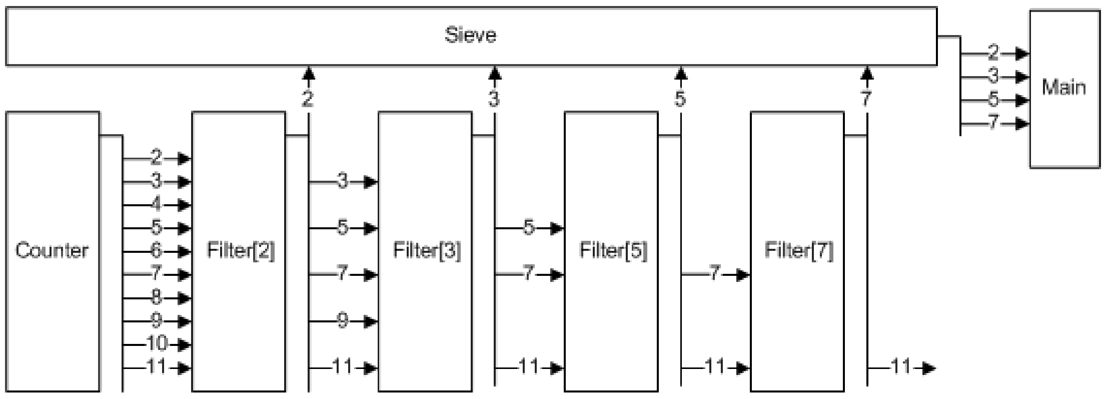
The sieve needs to first create the counter process, then successively wire it to each new filter process when it is created. The next prime number is the first number pulled out of the output channel of the counter/last filter.

The program's main method will print the first 100 primes that are generated by the sieve. The counter and filter processes have already been written and can be assumed to work correctly.

Your sieve method must have the signature: **`func sieve(prime chan int)`**. The parameter **`prime`** is the channel that new prime numbers should be pushed into. When the sieve reads the next prime number from the counter/filter chain, it should put it into the **`prime`** channel.

The counter function has the signature: **`func counter(count chan int)`**. The parameter **`count`** is the channel that generated numbers are written into. A counter will push a series of natural numbers starting at 2 and proceeding to infinity into the **`count`** channel. A counter is terminated when the program exits.

The filter function has the signature: **`func filter(prime int, recv chan int, send chan int)`**. The parameter **`prime`** is the number the filter should use for testing new numbers. The parameter **`recv`** is the channel that will be used to give numbers to the filter for testing. The parameter **`send`** is the channel that the filter will use to communicate all numbers that make it through. A filter pulls a number from its **`recv`** channel and, if it is not evenly divisible by **`prime`**, puts it into the **`send`** channel.


Sieve
Main
Counter
Filter[2]
Filter[3]
Filter[5]
Filter[7]
2
3
4
5
6
7
8
9
10
11

A.4 Coding Task - Actors Group

Time: 20 minutes

Write the `sieve` method for the actors implementation of the Sieve of Eratosthenes. Write your code in the file `prime_sieve.go` and compile it by running `make`. The file `prime.go` contains the remainder of the program and is required to build the program, but you are not allowed to look at its contents.

Expected Results

```
user@TIKI1 ~/coding $ ./prime
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191
193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283
293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401
409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509
521 523 541
```

Details

The sieve dedicates an actor to each filter, one actor for the counter (number generator), and one actor for the sieve. The sieve is the creator of the other actors. All actors communicate by requesting the next number from the "preceding actor" using a `("req", nil)` message and receive the answer as a `("num", n int)` message, where *n* is the next available number.

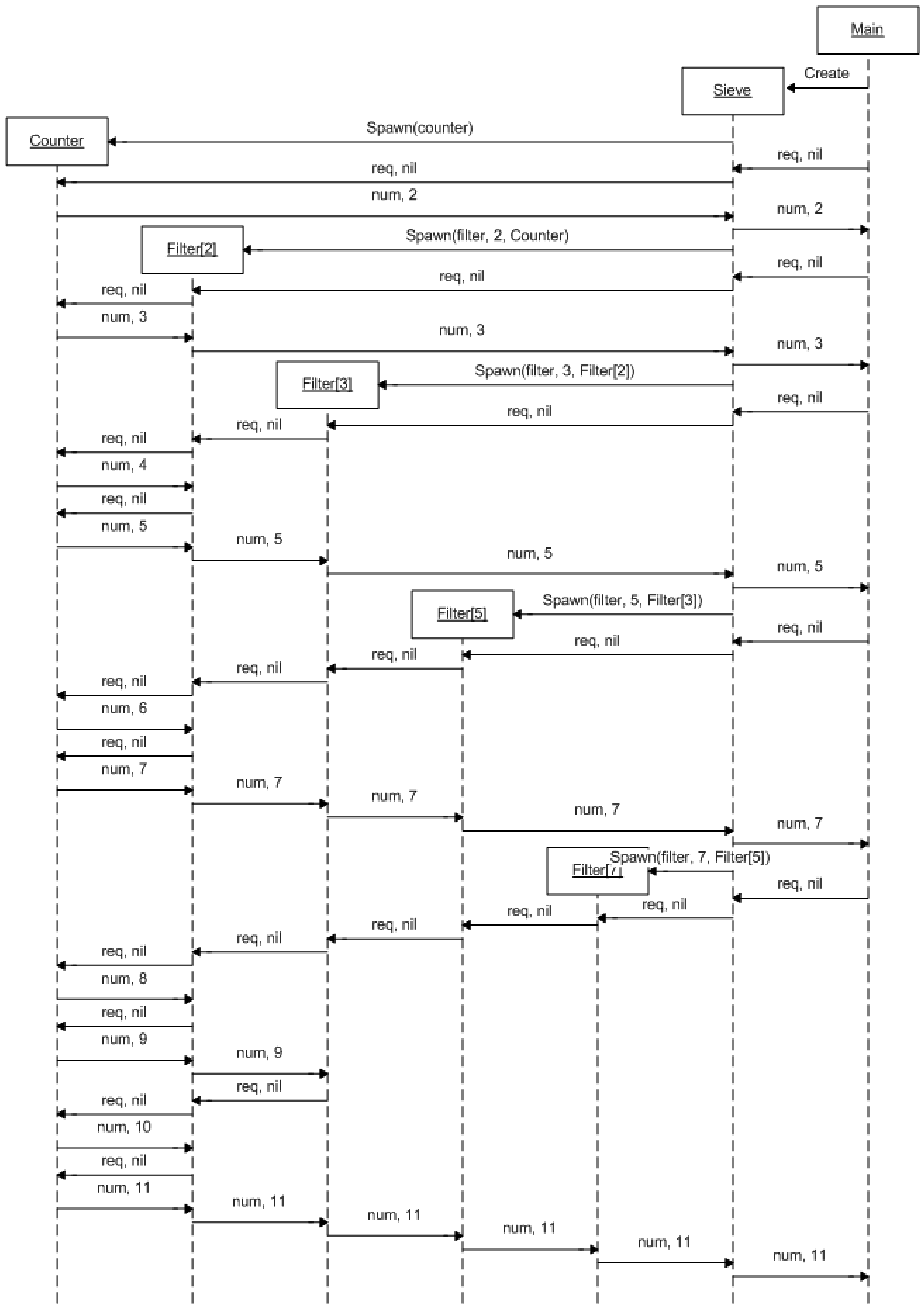
The sieve needs to first create the counter actor, then successively wire it to each new filter actor when it is created. The next prime number is the next number received after sending a request to the "previous" actor in the chain.

The program's main method will print the first 100 primes that are generated by the sieve. The counter and filter actors have already been written and can be assumed to work correctly.

Your sieve method must have the call signature `func sieve()` and the real signature `func sieve(a *action.Actor, args []interface{})`. The `args` variable will be empty. It will wait for a `("req", nil)` message, then return the next prime number to the sender of the "req" message as a `("num", n int)` message.

The counter method has the call signature: `func counter()`. It will wait for a `("req", nil)` message, then return the next countable number to the sender of the "req" message as a `("num", n int)` message.

The filter method has the call signature: `func filter(number int, in *action.ActorHandle)`. The `number` parameter is the number that the filter should filter by. The `in` parameter is the actor that the filter should request numbers from.


Main
Create
Sieve
Counter
Spawn(counter)
req, nil
num, 2
Filter[2]
Spawn(filter, 2, Counter)
num, 3
Filter[3]
Spawn(filter, 3, Filter[2])
num, 4
num, 5
Filter[5]
Spawn(filter, 5, Filter[3])
num, 6
num, 7
Filter[7]
Spawn(filter, 7, Filter[5])
num, 8
num, 9
num, 10
num, 11

Appendix B

Code

For the task code, the sections are ordered as: runner (black box code), file given to subject to start task, solution graded against (leniency allowed).

B.1 action.go

```
// Copyright (c) 2010, Chris Lieb
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions
// are met:
//
//     * Redistributions of source code must retain the above copyright
//       notice, this list of conditions and the following disclaimer.
//     * Redistributions in binary form must reproduce the above
//       copyright notice, this list of conditions and the following
//       disclaimer in the documentation and/or other materials provided
//       with the distribution.
//     * Neither the name of the Worcester Polytechnic Institute nor the
//       names of its contributors may be used to endorse or promote
//       products derived from this software without specific prior
//       written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
// INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
// BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
// OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
// AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
// LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY
// WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
// POSSIBILITY OF SUCH DAMAGE.

/*
    An implementation of the Actors programming model, based off of the
    actors APIs in Erlang and Scala.
*/
package action

import (
    "fmt"
    "os"
    "reflect"
    "time"
)

// A message that is sent from one Actor to another.  The Msg.Sender
// field can be used to reply to the message without needing prior
// knowledge of the sending Actor.
type Msg struct {
    // Handle to the actor that sent the message
    Sender *ActorHandle
    // Subject of the message. Used for pseudo-pattern-matching
    tag string
    // Body of the message
    Body interface{}
}

// The object passed to an actor to give them access to the
// functionality of an actor.
type Actor struct {
    // The pid of the Actor. Used for debugging
    pid int
    // The channel used for receiving messages
    rcvChan chan *Msg
    // The channel used to signal that this actor is finished
    finishChan chan bool
}
```

```
// A handle to an Actor created by the action.Spawn method.
type ActorHandle struct {
    // The actor that this handle is for
    actor *Actor
}

var (
    // registry of all running Actors
    pidRegistry map[int]*Actor
    // the next PID to issue
    nextPid int
)

// Initialize the package variables.  Automatically called when package
// is loaded for the first time.
func init() {
    nextPid = 1
    pidRegistry = make(map[int]*Actor)
}

// Starts an Actor-based program.  The function that is passed to Run is
// the "main" actor.  The args passed to Run are passed to the function
// when it is launched.
func Run(f func(*Actor, []interface{}), args ...interface{}) {
    // create the new actor object
    actor := new(Actor)
    actor.pid = nextPid
    nextPid++
    actor.rcvChan = make(chan *Msg, 100)
    actor.finishChan = make(chan bool, 1)
    pidRegistry[actor.pid] = actor

    // launch the new actor
    go f(actor, args)
}

// Spawn a new Actor.  The function passed is the code for the
// Actor and the args are the arguments passed to the Actor when it is
// launched.  A handle to the launched Actor is returned which can be
// used to send messages to the new Actor.
func Spawn(f func(*Actor, []interface{}),
                                args ...interface{}) *ActorHandle {
    // create the new actor object
    actor := new(Actor)
    actor.pid = nextPid
    nextPid++
    actor.rcvChan = make(chan *Msg, 100)
    actor.finishChan = make(chan bool, 1)
    pidRegistry[actor.pid] = actor

    // spawn the actor
    go f(actor, args)

    // wrap a handle around the new actor
    handle := new(ActorHandle)
    handle.actor = actor

    // return the handle
    return handle
}

// Wait for all Actors to finish. This call will block execution in the
// current thread until all Actors have signaled that they are done.
// This should be used at the end of the main() method to ensure that
// all Actors complete before the program quits.  Without this call, the
// program can terminate while there are Actors still running.
func FinishAll() {
    for pid, actor := range pidRegistry {
        <-actor.finishChan
        // delete the actor so we don't wait for it again
        pidRegistry[pid] = actor, false
    }
}

// Wait for a message to be sent to you. The messages you are listening
// for are expressed in the actions map, which maps message tags
// (subjects) to functions that handle those types of messages. If a
// message is sent to you that you are not listening for, it is dropped.
func (a *Actor) Receive(actions map[string]func(*Msg)) {
    // wait until we get a message that we can handle
    for {
        // force goroutine to sleep every time it checks for message to
        // make it give up the processor
        time.Sleep(1)

        msg := <-a.rcvChan
        f, ok := actions[msg.tag]
        if ok {
            f(msg)
            return
        } else {
            // recycle the unread message so it does not disappear
            a.rcvChan <- msg
        }
    }
}

// Wait for a messate to be sent to you.  If a message is not received
// in the given amount of time, execute a different function.  The
// messages you are listening for are expressed in the actions map,
// which maps message tags (subjects) to functions that handle those
// types of messages. If a message is sent to you that you are not
// listening for, it is dropped.
func (a *Actor) ReceiveTO(actions map[string]func(*Msg), delay int64,
        timeoutAction func(*Msg)) {
    // duplicate the given actions map since we hold a reference to it
    newActions := make(map[string]func(*Msg))
    for tag, action := range actions {
        newActions[tag] = action
    }

    // set the new timeout action
    newActions["action.timeout"] = timeoutAction

    // start the countdown timer
    Spawn(timeout, a, delay)

    // do a normal receive
    a.Receive(newActions)
}

// Wait a given amount of time (in milliseconds), then notify a
// listening actor.
//
// args:
//  a the Actor object
//  args [notify *Actor, delay int]
func timeout(a *Actor, args []interface{}) {
    var notify *Actor
    var delay int64
    var ok bool

    notify, ok = args[0].(*Actor)
    if !ok {
     fmt.Printf("First argument to timeout (nofity) should be *Actor\n")
```

```
        os.Exit(1)
    }
    delay, ok = args[1].(int64)
    if !ok {
       fmt.Printf("Second argument to timeout (delay) should be int\n")
        os.Exit(1)
    }

    time.Sleep(delay * int64(1000000))

    // wrap a handle around the new actor
    handle := new(ActorHandle)
    handle.actor = notify
    handle.Send(a, "action.timeout", nil)

    a.Done()
}

// Get a handle to this actor.
func (a *Actor) Handle() *ActorHandle {
    handle := new(ActorHandle)
    handle.actor = a
    return handle
}

// Signal that an Actor is finished.  Like a return statement in concept.
func (a *Actor) Done() {
    a.finishChan <- true
}

// Send a message to another actor.  First argument is yourself, so that
// the Actor receiving this message can reply to you.  The second arg
// is the tag (subject) of the message.  The third arg is the message
// body (payload).
func (h *ActorHandle) Send(a *Actor, tag string, msg interface{}) {
    message := new(Msg)
    handle := new(ActorHandle)
    handle.actor = a
    message.Sender = handle
    message.tag = tag
    message.Body = msg

    h.actor.rcvChan <- message
}
```

```
// Override the default toString method for structs
func (a *Actor) String() string {
    return fmt.Sprintf("Actor (PID %v)", a.pid)
}

// Override the default toString method for structs
func (a *ActorHandle) String() string {
    return fmt.Sprintf("Actor (PID %v)", a.actor.pid)
}

// Convert an array of interface{} values into type-casted forms
// provided as pointers.
//
// example:
//   args := []interface{}{5, "test"}
//   var t1 int
//   var t2 string
//   action.InterpretArgs(args, &t1, &t2)
//   fmt.Printf("t1 = %v, t2 = %v\n", t1, t2)
//    -> t1 = 5, t2 = test
func InterpretArgs(args []interface{}, vars ...interface{}) {
    if len(args) != len(vars) {
      fmt.Printf("!! Runtime error: args and vars must be same length " +
                             "(len(args) = %v, " + "len(vars) = %v)\n",
                                    len(args), len(vars))
        os.Exit(1)
    }

    for i := 0; i < len(args); i++ {
        v, ok := reflect.NewValue(vars[i]).(*reflect.PtrValue)
        if !ok {
           fmt.Printf("!! Runtime error: Out parameter %v must be a " +
                                    "pointer (given %v)\n", i+1,
                                    reflect.Typeof(vars[i]))
            os.Exit(1)
        }

        if v.Elem().Type() != reflect.Typeof(args[i]) {
         fmt.Printf("!! Runtime error: Argument %v has incorrect type " +
                                    "(expected %v, given %v)\n", i+1,
                             v.Elem().Type(), reflect.Typeof(args[i]))
            os.Exit(1)
        }

        v.Elem().SetValue(reflect.NewValue(args[i]))
    }
}
```

B.2 Baseline Task Code

B.2.1 bench_sort_test.go

```
package bench_sort;

import (
    "testing"
)

type SortStruct struct {
    in []int
    exp []int
}

func (s *SortStruct) equal() bool {
    if len(s.in) != len(s.exp) || cap(s.in) != cap(s.exp) {
        return false
    }

    for i := 0; i < len(s.in); i++ {
        if s.in[i] != s.exp[i] {
            return false
        }
    }

    return true
}

var SortTests = []SortStruct{
    SortStruct{
        []int{5},
        []int{5},
    },
    SortStruct{
        []int{4, 5},
        []int{4, 5},
    },
    SortStruct{
        []int{5, 4},
        []int{4, 5},
    },
    SortStruct{
        []int{5, 5},
        []int{5, 5},
    },
    SortStruct{
        []int{5, -5},
        []int {-5, 5},
    },
    SortStruct{
        []int{5, 4, -25, -42, 0, 1000, 1, -25, 90, 90},
        []int{-42, -25, -25, 0, 1, 4, 5, 90, 90, 1000},
    },
}

func TestAll(t *testing.T) {
    for _, r := range SortTests {
        Sort(r.in, Comp)
        if !r.equal() {
            t.Errorf("Expected %v, got %v", r.exp, r.in)
        }
    }
}
```

B.2.2 bench_sort.go.init

```
package bench_sort;
```

B.2.3 bench_sort.go

```
package bench_sort;

// Grading
// +1 - correctly declared function
// +1 - correctly implement outer loop
// +1 - correctly implement inner loop
// +1 - correctly compare array elements
// +1 - correctly swap array elements

func Sort(lst []int, comp func(obj1 int, obj2 int) int) {
    n := len(lst) - 1
    for i := 0; i <= n; i++ {
        for j := n; j > i; j-- {
            if comp(lst[j-1], lst[j]) > 0 {
                lst[j-1], lst[j] = lst[j], lst[j-1]
            }
        }
    }
}

// Grading
// +1 - correctly declared function
// +1 - returns correct values

func Comp(obj1 int, obj2 int) int {
    if (obj1 < obj2) {
        return -1
    } else if (obj1 > obj2) {
        return 1
    }
    return 0
}
```

B.3 Concurrent Coding Task Code - Channels

B.3.1 prime.go

```
package main;

import (
    "fmt"
)

func main() {
    prime := make(chan int, 0)
    go sieve(prime)
    for i := 0; i < 100; i++ {
        fmt.Printf("%v ", <-prime)
    }
    fmt.Printf("\n")
}

func counter(count chan int) {
    i := 1
    for {
        i++
        count <- i
    }
}
```

```
func filter(prime int, recv chan int, send chan int) {
    for {
        i := <-recv
        if (i % prime != 0) {
            send <- i
        }
    }
}
```

B.3.2 prime_sieve.go.init

```
package main;

func sieve(prime chan int) {

}
```

B.3.3 prime_sieve.go

```
package main;

// Grading
// +1 - create the counter process
// +1 - get a number from the counter/filter process
// +1 - send new prime
// +1 - create new filter process
// +1 - manage the counter and filter processes

func sieve(prime chan int) {
    c := make(chan int, 0)
    go counter(c)
    for {
        p := <-c
        prime <- p
        newc := make(chan int, 0)
        go filter(p, c, newc)
        c = newc
    }
}
```

B.4 Concurrent Coding Task Code - Actors

B.4.1 prime.go

```
package main;

import (
    "action"
    "fmt"
)

func main() {
    action.Run(actorMain)
    action.FinishAll()
}

func actorMain(a *action.Actor, args []interface{}) {
    p := action.Spawn(sieve)

    for i := 0; i < 100; i++ {
        p.Send(a, "req", nil)
        a.Receive(map[string]func(*action.Msg) {
            "num": func(msg *action.Msg) {
                fmt.Printf("%v ", msg.Body)
            },
        })
    }
    fmt.Printf("\n")
    p.Send(a, "quit", nil)
    a.Done()
    return
}

func counter(a *action.Actor, args []interface{}) {
    i := 1
    for {
        i++
        a.Receive(map[string]func(*action.Msg){
            "req": func(msg *action.Msg) {
                msg.Sender.Send(a, "num", i)
            },
            "quit": func(msg *action.Msg) {
                a.Done()
                return
            },
        })
    }
}

func filter(a *action.Actor, args []interface{}) {
    var number int
    var in *action.ActorHandle

    action.InterpretArgs(args, &number, &in)

    var newNum int
    for {
        a.Receive(map[string]func(*action.Msg) {
            "req": func(msg *action.Msg) {
                done := false
                for !done {
                    in.Send(a, "req", nil)
                    a.Receive(map[string]func(*action.Msg){
                        "num": func(msg2 *action.Msg) {
                            newNum = msg2.Body.(int)
                            if (newNum % number != 0) {
                                done = true
                                msg.Sender.Send(a, "num", newNum)
                            }
                        },
                        "quit": func(msg *action.Msg) {
                            in.Send(a, "quit", nil)
                            a.Done()
                            return
                        },
                    })
                }
            },
            "quit": func(msg *action.Msg) {
                in.Send(a, "quit", nil)
                a.Done()
                return
            },
        })
    }
}
```

B.4.2 prime_sieve.go.init

```
package main;
```

```
import (
    "action"
)

func sieve(a *action.Actor, args []interface{}) {

}
```

B.4.3 prime_sieve.go

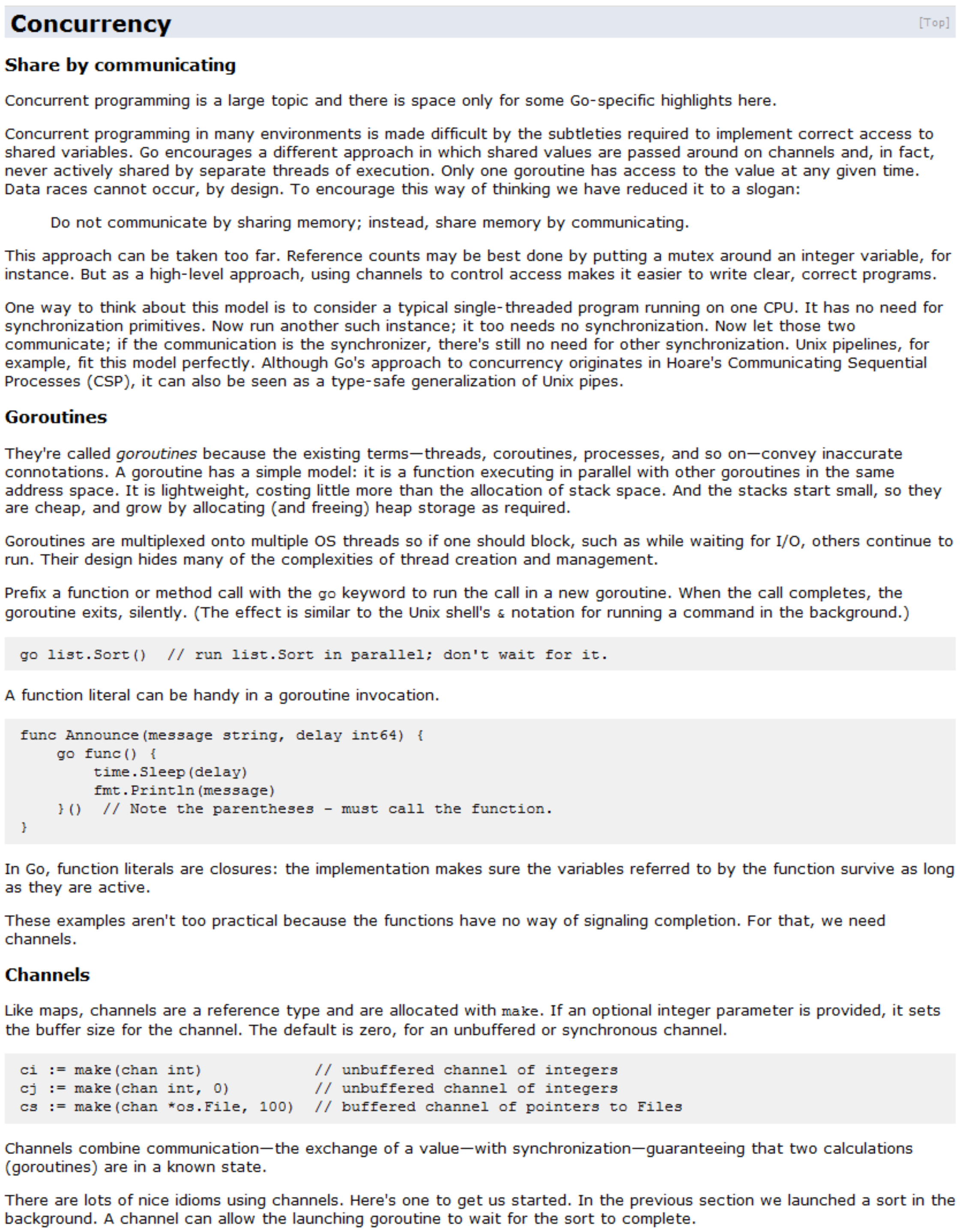
```
package main;

import (
    "action"
)

// Grading
// +1 - create the counter process
// +1 - get a number from the counter/filter process
// +1 - send new prime
// +1 - create new filter process
// +1 - manage the counter and filter processes

func sieve(a *action.Actor, args []interface{}) {
    last := action.Spawn(counter)
    for {
        var num int
        last.Send(a, "req", nil)
        a.Receive(map[string]func(*action.Msg){
            "num": func(msg *action.Msg) {
                num = msg.Body.(int)
                a.Receive(map[string]func(*action.Msg){
                    "req": func(msg *action.Msg) {
                        msg.Sender.Send(a, "num", num)
                    },
                    "quit": func(msg *action.Msg) {
                        last.Send(a, "quit", nil)
                        a.Done()
                        return
                    },
                })
                last = action.Spawn(filter, num, last)
            },
            "quit": func(msg *action.Msg) {
                last.Send(a, "quit", nil)
                a.Done()
                return
            },
        })
    }
}
```

Appendix C

Documentation

C.1 Effective Go - Channels Section

This section of the Effective Go tutorial was seen only by the channels group. It is part of the actual Effective Go tutorial that can be found at http://golang.org/doc/effective_go.html



Concurrency

[Top]

Share by communicating

Concurrent programming is a large topic and there is space only for some Go-specific highlights here.

Concurrent programming in many environments is made difficult by the subtleties required to implement correct access to shared variables. Go encourages a different approach in which shared values are passed around on channels and, in fact, never actively shared by separate threads of execution. Only one goroutine has access to the value at any given time. Data races cannot occur, by design. To encourage this way of thinking we have reduced it to a slogan:

> Do not communicate by sharing memory; instead, share memory by communicating.

This approach can be taken too far. Reference counts may be best done by putting a mutex around an integer variable, for instance. But as a high-level approach, using channels to control access makes it easier to write clear, correct programs.

One way to think about this model is to consider a typical single-threaded program running on one CPU. It has no need for synchronization primitives. Now run another such instance; it too needs no synchronization. Now let those two communicate; if the communication is the synchronizer, there's still no need for other synchronization. Unix pipelines, for example, fit this model perfectly. Although Go's approach to concurrency originates in Hoare's Communicating Sequential Processes (CSP), it can also be seen as a type-safe generalization of Unix pipes.

Goroutines

They're called *goroutines* because the existing terms—threads, coroutines, processes, and so on—convey inaccurate connotations. A goroutine has a simple model: it is a function executing in parallel with other goroutines in the same address space. It is lightweight, costing little more than the allocation of stack space. And the stacks start small, so they are cheap, and grow by allocating (and freeing) heap storage as required.

Goroutines are multiplexed onto multiple OS threads so if one should block, such as while waiting for I/O, others continue to run. Their design hides many of the complexities of thread creation and management.

Prefix a function or method call with the `go` keyword to run the call in a new goroutine. When the call completes, the goroutine exits, silently. (The effect is similar to the Unix shell's `&` notation for running a command in the background.)

```
go list.Sort()  // run list.Sort in parallel; don't wait for it.
```

A function literal can be handy in a goroutine invocation.

```
func Announce(message string, delay int64) {
    go func() {
        time.Sleep(delay)
        fmt.Println(message)
    }()  // Note the parentheses - must call the function.
}
```

In Go, function literals are closures: the implementation makes sure the variables referred to by the function survive as long as they are active.

These examples aren't too practical because the functions have no way of signaling completion. For that, we need channels.

Channels

Like maps, channels are a reference type and are allocated with `make`. If an optional integer parameter is provided, it sets the buffer size for the channel. The default is zero, for an unbuffered or synchronous channel.

```
ci := make(chan int)            // unbuffered channel of integers
cj := make(chan int, 0)         // unbuffered channel of integers
cs := make(chan *os.File, 100)  // buffered channel of pointers to Files
```

Channels combine communication—the exchange of a value—with synchronization—guaranteeing that two calculations (goroutines) are in a known state.

There are lots of nice idioms using channels. Here's one to get us started. In the previous section we launched a sort in the background. A channel can allow the launching goroutine to wait for the sort to complete.

```
c := make(chan int)  // Allocate a channel.
// Start the sort in a goroutine; when it completes, signal on the channel.
go func() {
    list.Sort()
    c <- 1  // Send a signal; value does not matter.
}()
doSomethingForAWhile()
<-c   // Wait for sort to finish; discard sent value.
```

Receivers always block until there is data to receive. If the channel is unbuffered, the sender blocks until the receiver has received the value. If the channel has a buffer, the sender blocks only until the value has been copied to the buffer; if the buffer is full, this means waiting until some receiver has retrieved a value.

A buffered channel can be used like a semaphore, for instance to limit throughput. In this example, incoming requests are passed to `handle`, which sends a value into the channel, processes the request, and then receives a value from the channel. The capacity of the channel buffer limits the number of simultaneous calls to `process`.

```
var sem = make(chan int, MaxOutstanding)

func handle(r *Request) {
    sem <- 1    // Wait for active queue to drain.
    process(r)  // May take a long time.
    <-sem       // Done; enable next request to run.
}

func Serve(queue chan *Request) {
    for {
        req := <-queue
        go handle(req)  // Don't wait for handle to finish.
    }
}
```

Here's the same idea implemented by starting a fixed number of `handle` goroutines all reading from the request channel. The number of goroutines limits the number of simultaneous calls to `process`. This `Serve` function also accepts a channel on which it will be told to exit; after launching the goroutines it blocks receiving from that channel.

```
func handle(queue chan *Request) {
    for r := range queue {
        process(r)
    }
}

func Serve(clientRequests chan *clientRequests, quit chan bool) {
    // Start handlers
    for i := 0; i < MaxOutstanding; i++ {
        go handle(clientRequests)
    }
    <-quit  // Wait to be told to exit.
}
```

Channels of channels

One of the most important properties of Go is that a channel is a first-class value that can be allocated and passed around like any other. A common use of this property is to implement safe, parallel demultiplexing.

In the example in the previous section, `handle` was an idealized handler for a request but we didn't define the type it was handling. If that type includes a channel on which to reply, each client can provide its own path for the answer. Here's a schematic definition of type `Request`.

```
type Request struct {
    args        []int
    f           func([]int) int
    resultChan  chan int
}
```

The client provides a function and its arguments, as well as a channel inside the request object on which to receive the answer.

```
func sum(a []int) (s int) {
    for _, v := range a {
        s += v
    }
    return
}

request := &Request{[]int{3, 4, 5}, sum, make(chan int)}
// Send request
clientRequests <- request
// Wait for response.
fmt.Printf("answer: %d\n", <-request.resultChan)
```

On the server side, the handler function is the only thing that changes.

```
func handle(queue chan *Request) {
    for req := range queue {
        req.resultChan <- req.f(req.args)
    }
}
```

There's clearly a lot more to do to make it realistic, but this code is a framework for a rate-limited, parallel, non-blocking RPC system, and there's not a mutex in sight.

Parallelization

Another application of these ideas is to parallelize a calculation across multiple CPU cores. If the calculation can be broken into separate pieces, it can be parallelized, with a channel to signal when each piece completes.

Let's say we have an expensive operation to perform on a vector of items, and that the value of the operation on each item is independent, as in this idealized example.

```
type Vector []float64

// Apply the operation to v[i], v[i+1] ... up to v[n-1].
func (v Vector) DoSome(i, n int, u Vector, c chan int) {
    for ; i < n; i++ {
        v[i] += u.Op(v[i])
    }
    c <- 1    // signal that this piece is done
}
```

We launch the pieces independently in a loop, one per CPU. They can complete in any order but it doesn't matter; we just count the completion signals by draining the channel after launching all the goroutines.

```
const NCPU = 4  // number of CPU cores

func (v Vector) DoAll(u Vector) {
    c := make(chan int, NCPU)  // Buffering optional but sensible.
    for i := 0; i < NCPU; i++ {
        go v.DoSome(i*len(v)/NCPU, (i+1)*len(v)/NCPU, u, c)
    }
    // Drain the channel.
    for i := 0; i < NCPU; i++ {
        <-c    // wait for one task to complete
    }
    // All done.
}
```

The current implementation of `gc` (`6g`, etc.) will not parallelize this code by default. It dedicates only a single core to user-level processing. An arbitrary number of goroutines can be blocked in system calls, but by default only one can be executing user-level code at any time. It should be smarter and one day it will be smarter, but until it is if you want CPU parallelism you must tell the run-time how many goroutines you want executing code simultaneously. There are two related ways to do this. Either run your job with environment variable `GOMAXPROCS` set to the number of cores to use (default 1); or import the `runtime` package and call `runtime.GOMAXPROCS(NCPU)`. Again, this requirement is expected to be retired as the scheduling and run-time improve.

A leaky buffer

The tools of concurrent programming can even make non-concurrent ideas easier to express. Here's an example abstracted from an RPC package. The client goroutine loops receiving data from some source, perhaps a network. To avoid allocating and freeing buffers, it keeps a free list, and uses a buffered channel to represent it. If the channel is empty, a new buffer gets allocated. Once the message buffer is ready, it's sent to the server on `serverChan`.

```
var freeList = make(chan *Buffer, 100)
var serverChan = make(chan *Buffer)

func client() {
    for {
        var b *Buffer
        // Grab a buffer if available; allocate if not.
        select {
        case b = <-freeList:
            // Got one; nothing more to do.
        default:
            // None free, so allocate a new one.
            b = new(Buffer)
        }
        load(b)              // Read next message from the net.
        serverChan <- b      // Send to server.
    }
}
```

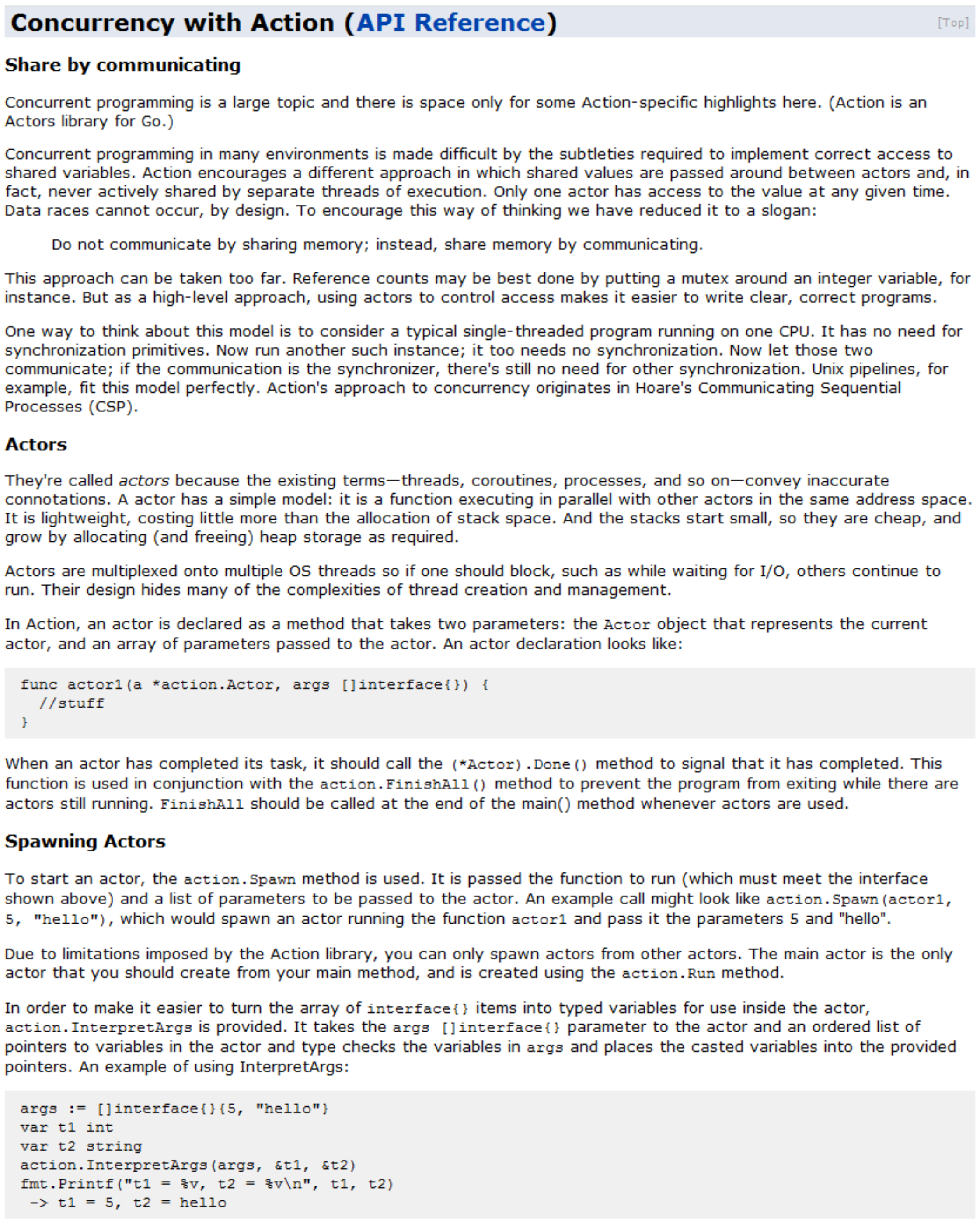
The server loop receives each message from the client, processes it, and returns the buffer to the free list.

```
func server() {
    for {
        b := <-serverChan    // Wait for work.
        process(b)
        // Reuse buffer if there's room.
        select {
        case freeList <- b:
            // Buffer on free list; nothing more to do.
        default:
            // Free list full, just carry on.
        }
    }
}
```

The client attempts to retrieve a buffer from `freeList`; if none is available, it allocates a fresh one. The server's send to `freeList` puts `b` back on the free list unless the list is full, in which case the buffer is dropped on the floor to be reclaimed by the garbage collector. (The `default` clauses in the `select` statements execute when no other case is ready, meaning that the `selects` never block.) This implementation builds a leaky bucket free list in just a few lines, relying on the buffered channel and the garbage collector for bookkeeping.

C.2 Effective Go - Actors Section

This section of the Effective Go tutorial was seen only by the actors group. It was created by the author and substituted into the Effective Go tutorial for the section on channels.



Concurrency with Action (API Reference)

[Top]

Share by communicating

Concurrent programming is a large topic and there is space only for some Action-specific highlights here. (Action is an Actors library for Go.)

Concurrent programming in many environments is made difficult by the subtleties required to implement correct access to shared variables. Action encourages a different approach in which shared values are passed around between actors and, in fact, never actively shared by separate threads of execution. Only one actor has access to the value at any given time. Data races cannot occur, by design. To encourage this way of thinking we have reduced it to a slogan:

> Do not communicate by sharing memory; instead, share memory by communicating.

This approach can be taken too far. Reference counts may be best done by putting a mutex around an integer variable, for instance. But as a high-level approach, using actors to control access makes it easier to write clear, correct programs.

One way to think about this model is to consider a typical single-threaded program running on one CPU. It has no need for synchronization primitives. Now run another such instance; it too needs no synchronization. Now let those two communicate; if the communication is the synchronizer, there's still no need for other synchronization. Unix pipelines, for example, fit this model perfectly. Action's approach to concurrency originates in Hoare's Communicating Sequential Processes (CSP).

Actors

They're called *actors* because the existing terms—threads, coroutines, processes, and so on—convey inaccurate connotations. A actor has a simple model: it is a function executing in parallel with other actors in the same address space. It is lightweight, costing little more than the allocation of stack space. And the stacks start small, so they are cheap, and grow by allocating (and freeing) heap storage as required.

Actors are multiplexed onto multiple OS threads so if one should block, such as while waiting for I/O, others continue to run. Their design hides many of the complexities of thread creation and management.

In Action, an actor is declared as a method that takes two parameters: the `Actor` object that represents the current actor, and an array of parameters passed to the actor. An actor declaration looks like:

```
func actor1(a *action.Actor, args []interface{}) {
  //stuff
}
```

When an actor has completed its task, it should call the `(*Actor).Done()` method to signal that it has completed. This function is used in conjunction with the `action.FinishAll()` method to prevent the program from exiting while there are actors still running. `FinishAll` should be called at the end of the main() method whenever actors are used.

Spawning Actors

To start an actor, the `action.Spawn` method is used. It is passed the function to run (which must meet the interface shown above) and a list of parameters to be passed to the actor. An example call might look like `action.Spawn(actor1, 5, "hello")`, which would spawn an actor running the function `actor1` and pass it the parameters 5 and "hello".

Due to limitations imposed by the Action library, you can only spawn actors from other actors. The main actor is the only actor that you should create from your main method, and is created using the `action.Run` method.

In order to make it easier to turn the array of `interface{}` items into typed variables for use inside the actor, `action.InterpretArgs` is provided. It takes the `args []interface{}` parameter to the actor and an ordered list of pointers to variables in the actor and type checks the variables in `args` and places the casted variables into the provided pointers. An example of using InterpretArgs:

```
args := []interface{}{5, "hello"}
var t1 int
var t2 string
action.InterpretArgs(args, &t1, &t2)
fmt.Printf("t1 = %v, t2 = %v\n", t1, t2)
 -> t1 = 5, t2 = hello
```

Communication

Each actor has a single input communication channel that is used to send messages to it. Multiple actors can send messages into this channel, but only the actor that is belongs to can read data out of it.

Messages take the form of a sender, a tag (subject), and a body. The sender is the `Actor` object of the actor that is sending the message, and is used for replying to the sender of the message. The tag is a string that describes what the message is about, and is used by the receiving actor to decide what to do with the message. The body is any type of value that you want to pass as part of the message. If the message does not have a need for a body, simply use the value `nil` in place of the message.

When an actor is created using the `Spawn` method, an `action.ActorHandle` is returned that is used to send messages to that actor. This is done using the `ActorHandle.Send` method, which takes an `Actor` object that represents the sender of the message, a subject tag, and the body of the message. The message is sent to the actor represented by the `ActorHandle` asynchronously, allowing the sending actor to continue working without waiting for the receiving actor to receive the message.

```
h.Send(a, "test1", 5)
```

In order to receive a message, an actor uses either the `Actor.Receive` or `Actor.ReceiveTO` method. The `Receive` method takes a single parameter that is a map from strings to functions that take a parameter of a pointer to a message. Every message that is sent to an actor is placed into a queue, where it is held until the actor starts a Receive. When a receive is started, messages are pulled from the queue until one is found that has a tag matching one of the tags in the map passed to the Receive call. The function that is mapped to that tag is then called and passed the Msg object that was the matched message. If no message with a matching tag is found, the actor continues to wait for a message with a matching tag to be received.

```
a.Receive(map[string]func(*action.Msg){
        "test1": func(msg *action.Msg) {
                fmt.Printf("%+v\n", msg)
                msg.Sender.Send(a, "re", "test")
        }})
```

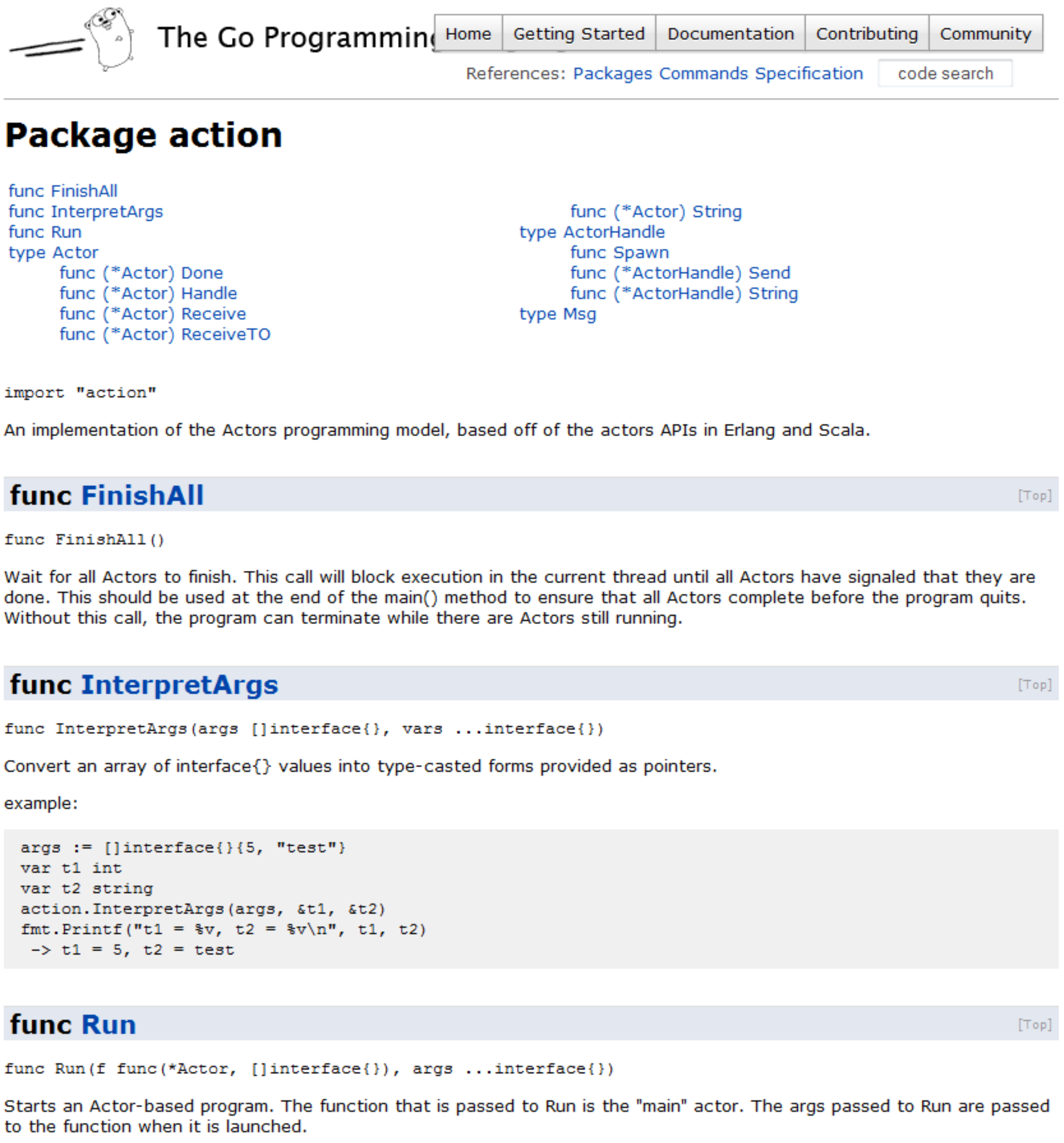
The `ReceiveTO` method acts just like the `Receive` method, except it only waits a given amount of time for a matching message to arrive. If no message arrives in the given timeframe, then a user-specified timeout funciton is called and the method exits.

```
a.ReceiveTO(map[string]func(*action.Msg){
        "re": func(msg *action.Msg) {
                fmt.Printf("%v\n", msg.Body)
        }},
        2000, func(msg *action.Msg) {
                fmt.Printf("timeout\n")
        })
```

Parallelization

The current implementation of `gc` (`6g`, etc.) will not parallelize this code by default. It dedicates only a single core to user-level processing. An arbitrary number of goroutines can be blocked in system calls, but by default only one can be executing user-level code at any time. It should be smarter and one day it will be smarter, but until it is if you want CPU parallelism you must tell the run-time how many goroutines you want executing code simultaneously. There are two related ways to do this. Either run your job with environment variable `GOMAXPROCS` set to the number of cores to use (default 1); or import the `runtime` package and call `runtime.GOMAXPROCS(NCPU)`. Again, this requirement is expected to be retired as the scheduling and run-time improve.

C.3 Action API



Package action

```
import "action"
```

An implementation of the Actors programming model, based off of the actors APIs in Erlang and Scala.

func FinishAll [Top]

```
func FinishAll()
```

Wait for all Actors to finish. This call will block execution in the current thread until all Actors have signaled that they are done. This should be used at the end of the main() method to ensure that all Actors complete before the program quits. Without this call, the program can terminate while there are Actors still running.

func InterpretArgs [Top]

```
func InterpretArgs(args []interface{}, vars ...interface{})
```

Convert an array of interface{} values into type-casted forms provided as pointers.

example:

```
args := []interface{}{5, "test"}
var t1 int
var t2 string
action.InterpretArgs(args, &t1, &t2)
fmt.Printf("t1 = %v, t2 = %v\n", t1, t2)
 -> t1 = 5, t2 = test
```

func Run [Top]

```
func Run(f func(*Actor, []interface{}), args ...interface{})
```

Starts an Actor-based program. The function that is passed to Run is the "main" actor. The args passed to Run are passed to the function when it is launched.

type Actor [Top]

The object passed to an actor to give them access to the functionality of an actor.

```
type Actor struct {
    // contains unexported fields
}
```

func (*Actor) Done

```
func (a *Actor) Done()
```

Signal that an Actor is finished. Like a return statement in concept.

func (*Actor) Handle

```
func (a *Actor) Handle() *ActorHandle
```

Get a handle to this actor.

func (*Actor) Receive

```
func (a *Actor) Receive(actions map[string]func(*Msg))
```

Wait for a message to be sent to you. The messages you are listening for are expressed in the actions map, which maps message tags (subjects) to functions that handle those types of messages. If a message is sent to you that you are not listening for, it is dropped.

func (*Actor) ReceiveTO

```
func (a *Actor) ReceiveTO(actions map[string]func(*Msg), delay int64, timeoutAction func(*Msg))
```

Wait for a messate to be sent to you. If a message is not received in the given amount of time, execute a different function. The messages you are listening for are expressed in the actions map, which maps message tags (subjects) to functions that handle those types of messages. If a message is sent to you that you are not listening for, it is dropped.

func (*Actor) String

```
func (a *Actor) String() string
```

Override the default toString method for structs

type ActorHandle [Top]

A handle to an Actor created by the action.Spawn method.

```
type ActorHandle struct {
    // contains unexported fields
}
```

func Spawn

```
func Spawn(f func(*Actor, []interface{}), args ...interface{}) *ActorHandle
```

Spawn a new Actor. The function passed is the code for the Actor and the args are the arguments passed to the Actor when it is launched. A handle to the launched Actor is returned which can be used to send messages to the new Actor.

func (*ActorHandle) Send

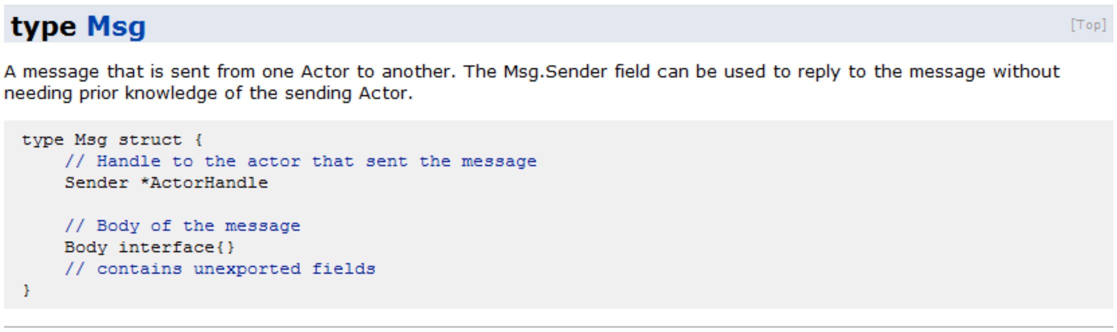
```
func (h *ActorHandle) Send(a *Actor, tag string, msg interface{})
```

Send a message to another actor. First argument is yourself, so that the Actor receiving this message can reply to you. The second arg is the tag (subject) of the message. The third arg is the message body (payload).

func (*ActorHandle) String

```
func (a *ActorHandle) String() string
```

Override the default toString method for structs



type Msg

[Top]

A message that is sent from one Actor to another. The Msg.Sender field can be used to reply to the message without needing prior knowledge of the sending Actor.

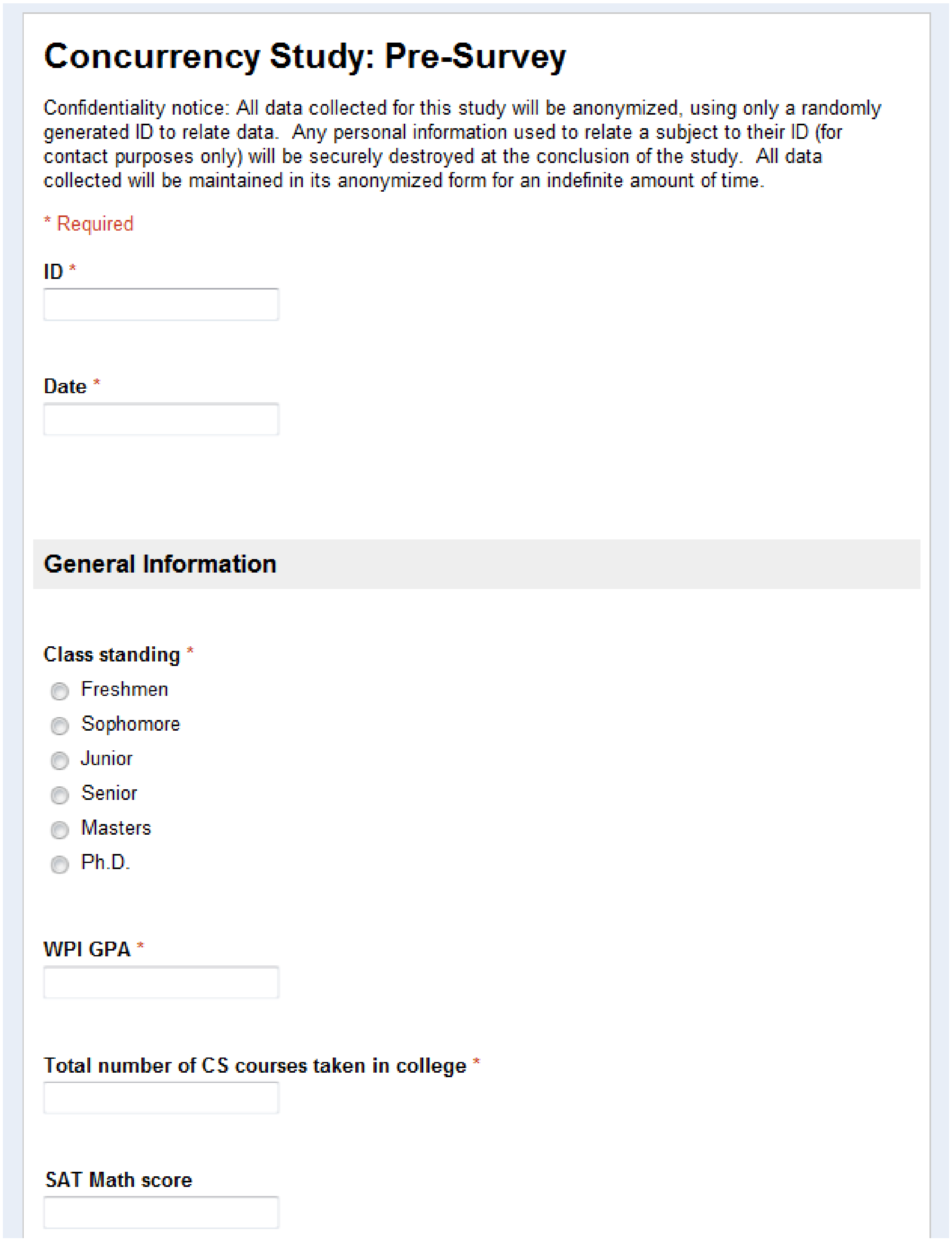
```
type Msg struct {
    // Handle to the actor that sent the message
    Sender *ActorHandle

    // Body of the message
    Body interface{}
    // contains unexported fields
}
```

Appendix D

Surveys

D.1 Pre-survey

This survey was filled out by subjects before they were assigned to a study group or given any material to study.



Concurrency Study: Pre-Survey

Confidentiality notice: All data collected for this study will be anonymized, using only a randomly generated ID to relate data. Any personal information used to relate a subject to their ID (for contact purposes only) will be securely destroyed at the conclusion of the study. All data collected will be maintained in its anonymized form for an indefinite amount of time.

* Required

ID *

Date *

General Information

Class standing *

- Freshmen
- Sophomore
- Junior
- Senior
- Masters
- Ph.D.

WPI GPA *

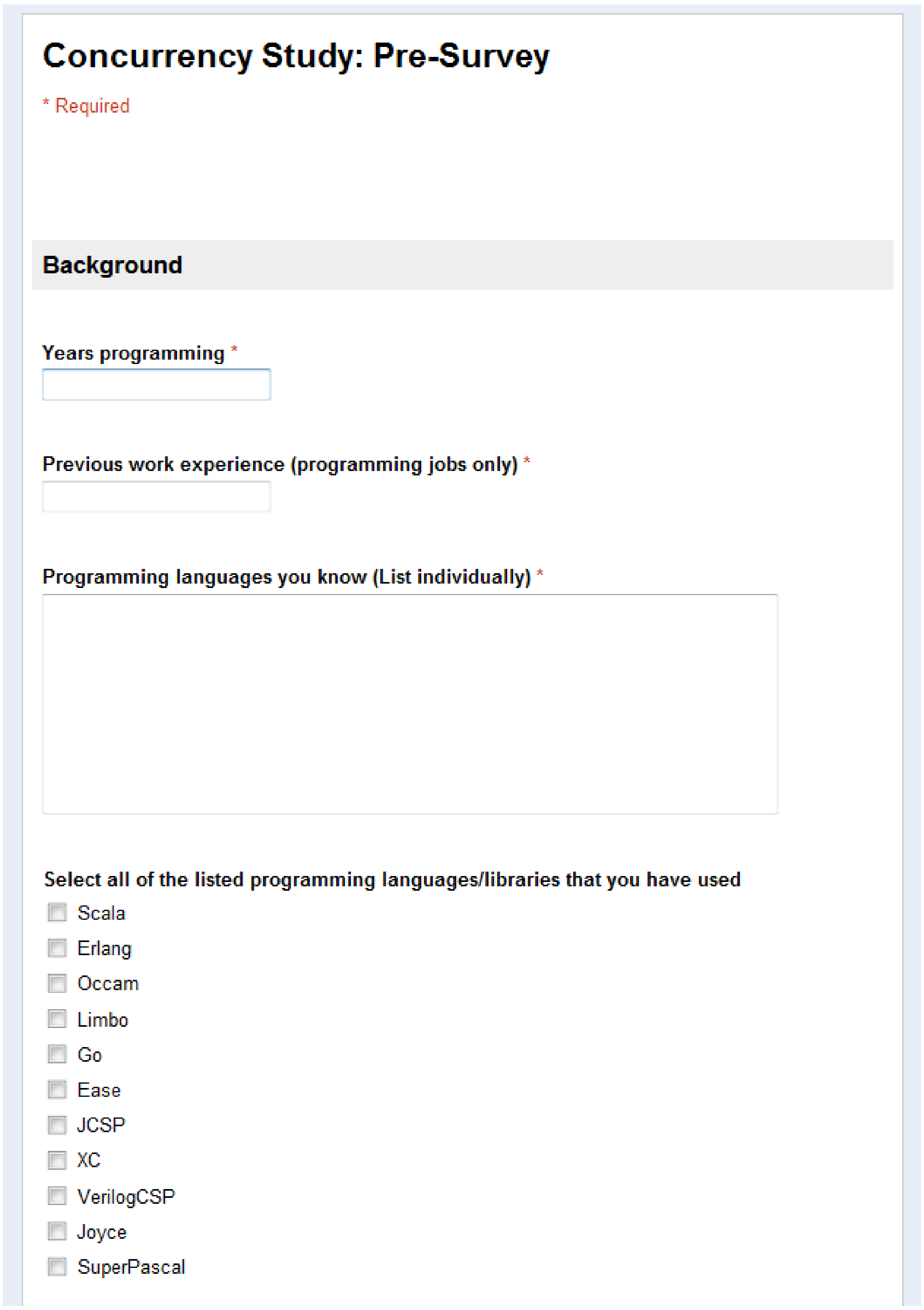
Total number of CS courses taken in college *

SAT Math score

GRE Math score

Continue »

Powered by Google Docs



Concurrency Study: Pre-Survey

* Required

Background

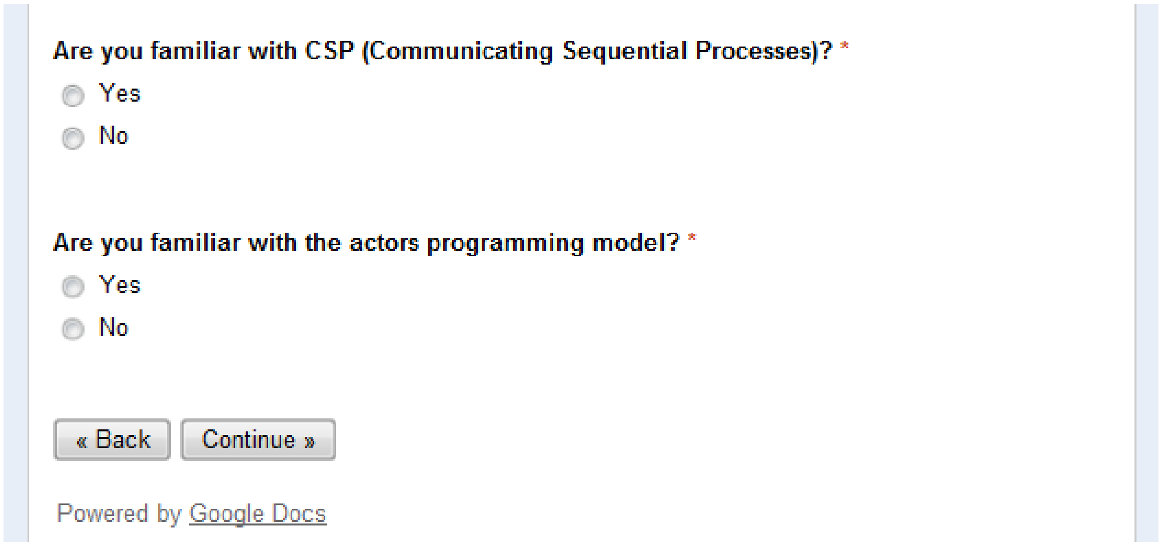
Years programming *

Previous work experience (programming jobs only) *

Programming languages you know (List individually) *

Select all of the listed programming languages/libraries that you have used

- ☐ Scala
- ☐ Erlang
- ☐ Occam
- ☐ Limbo
- ☐ Go
- ☐ Ease
- ☐ JCSP
- ☐ XC
- ☐ VerilogCSP
- ☐ Joyce
- ☐ SuperPascal


Are you familiar with CSP (Communicating Sequential Processes)? *

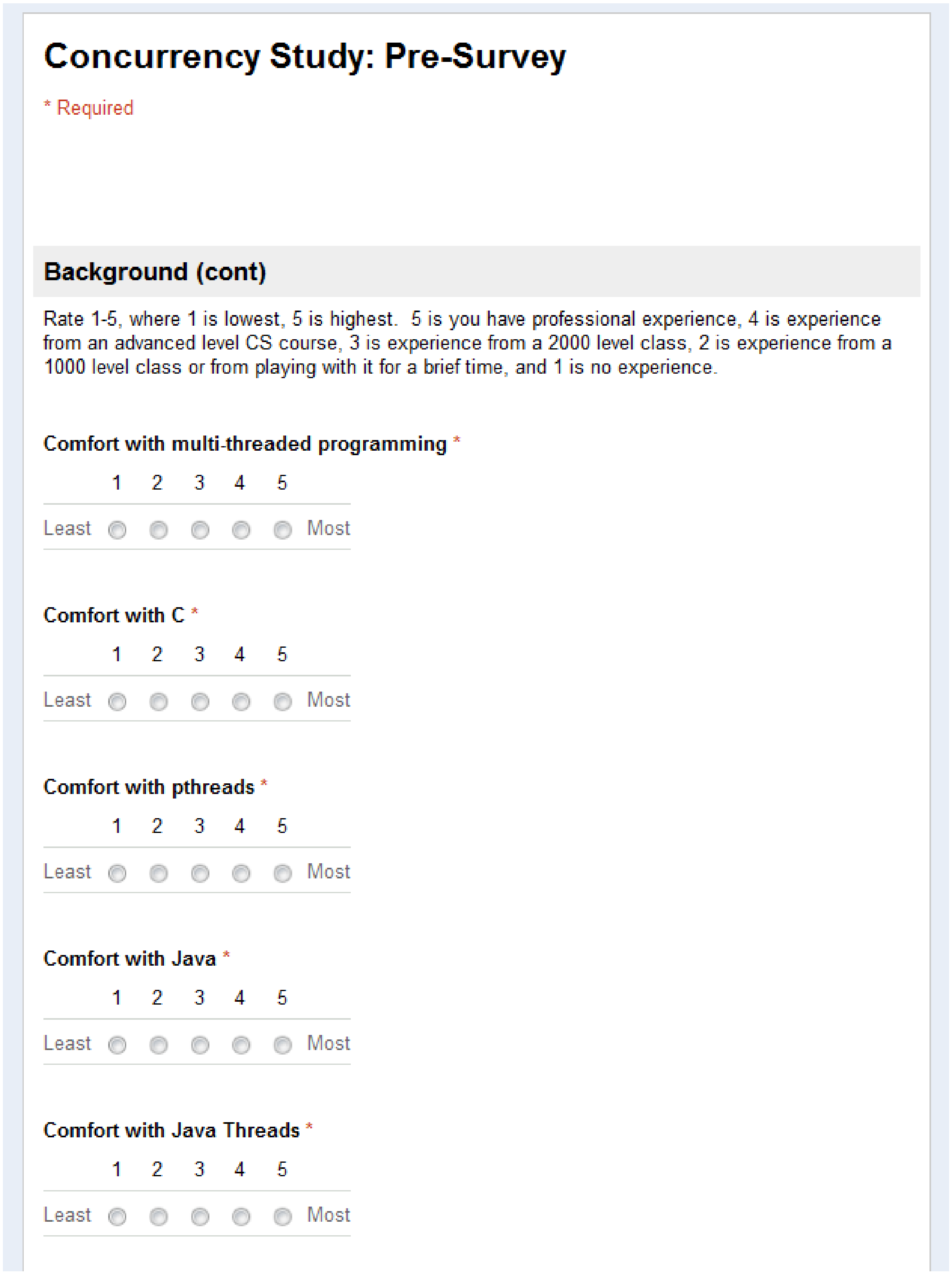
○ Yes

○ No

Are you familiar with the actors programming model? *

○ Yes

○ No



Concurrency Study: Pre-Survey

* Required

Background (cont)

Rate 1-5, where 1 is lowest, 5 is highest. 5 is you have professional experience, 4 is experience from an advanced level CS course, 3 is experience from a 2000 level class, 2 is experience from a 1000 level class or from playing with it for a brief time, and 1 is no experience.

Comfort with multi-threaded programming *

	1	2	3	4	5	
Least	○	○	○	○	○	Most

Comfort with C *

	1	2	3	4	5	
Least	○	○	○	○	○	Most

Comfort with pthreads *

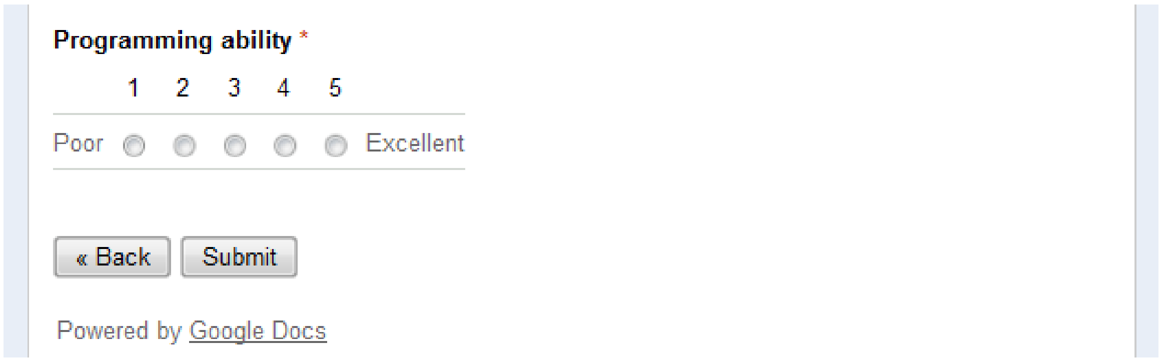
	1	2	3	4	5	
Least	○	○	○	○	○	Most

Comfort with Java *

	1	2	3	4	5	
Least	○	○	○	○	○	Most

Comfort with Java Threads *

	1	2	3	4	5	
Least	○	○	○	○	○	Most

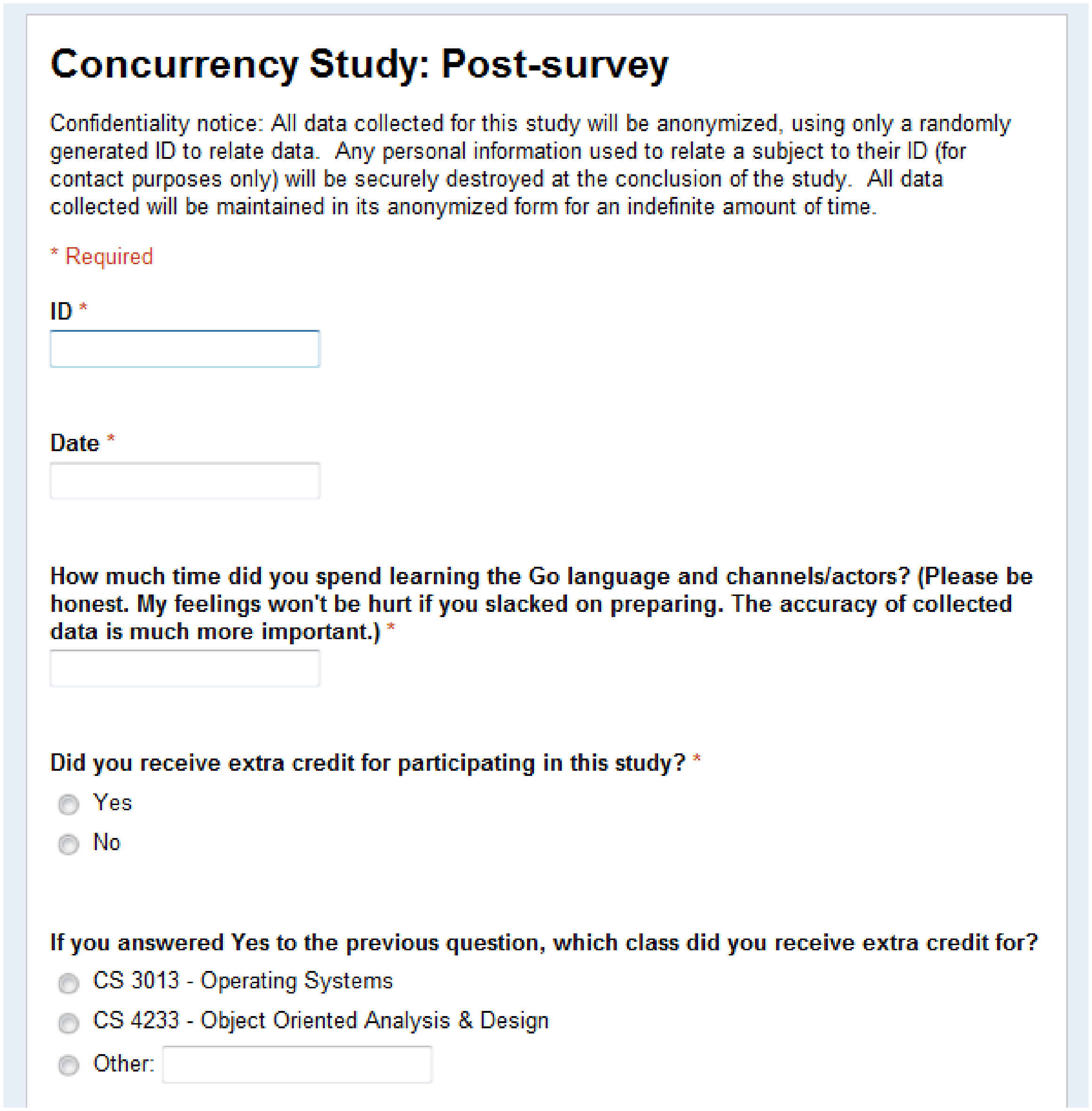


Programming ability *

	1	2	3	4	5	
Poor	○	○	○	○	○	Excellent

D.2 Post-survey

This survey was given to subjects after their concluding meeting with the author and marked the end of the subject's participation in the study.



Concurrency Study: Post-survey

Confidentiality notice: All data collected for this study will be anonymized, using only a randomly generated ID to relate data. Any personal information used to relate a subject to their ID (for contact purposes only) will be securely destroyed at the conclusion of the study. All data collected will be maintained in its anonymized form for an indefinite amount of time.

* Required

ID *

Date *

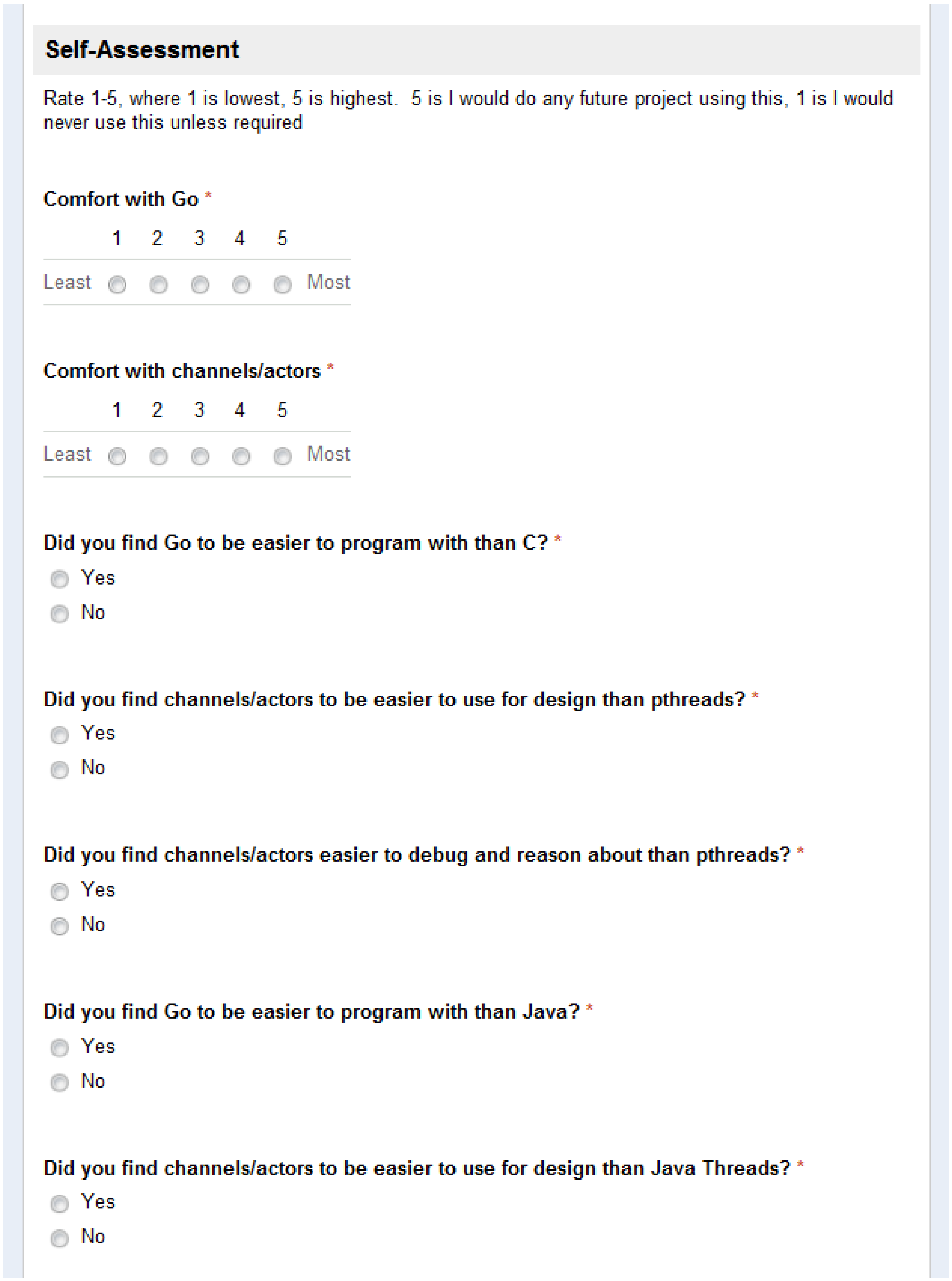
How much time did you spend learning the Go language and channels/actors? (Please be honest. My feelings won't be hurt if you slacked on preparing. The accuracy of collected data is much more important.) *

Did you receive extra credit for participating in this study? *

- Yes
- No

If you answered Yes to the previous question, which class did you receive extra credit for?

- CS 3013 - Operating Systems
- CS 4233 - Object Oriented Analysis & Design
- Other:



Self-Assessment

Rate 1-5, where 1 is lowest, 5 is highest. 5 is I would do any future project using this, 1 is I would never use this unless required

Comfort with Go *

	1	2	3	4	5	
Least	○	○	○	○	○	Most

Comfort with channels/actors *

	1	2	3	4	5	
Least	○	○	○	○	○	Most

Did you find Go to be easier to program with than C? *

○ Yes
○ No

Did you find channels/actors to be easier to use for design than pthreads? *

○ Yes
○ No

Did you find channels/actors easier to debug and reason about than pthreads? *

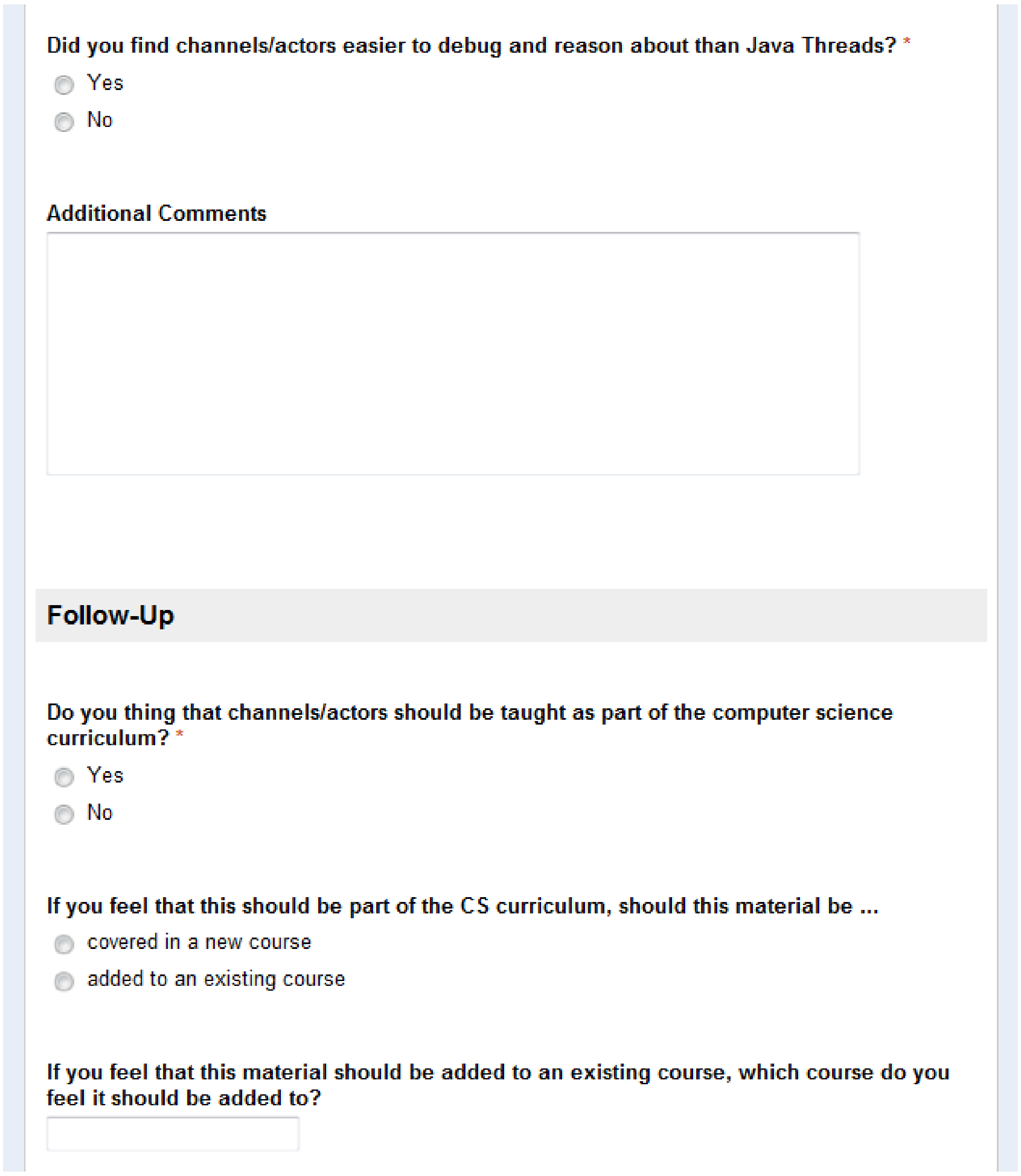
○ Yes
○ No

Did you find Go to be easier to program with than Java? *

○ Yes
○ No

Did you find channels/actors to be easier to use for design than Java Threads? *

○ Yes
○ No



Did you find channels/actors easier to debug and reason about than Java Threads? *

○ Yes

○ No

Additional Comments

Follow-Up

Do you thing that channels/actors should be taught as part of the computer science curriculum? *

○ Yes

○ No

If you feel that this should be part of the CS curriculum, should this material be ...

○ covered in a new course

○ added to an existing course

If you feel that this material should be added to an existing course, which course do you feel it should be added to?

Appendix E

Course Descriptions

CS 2303: Systems Programming Concepts

Cat. I

This course introduces students to a model of programming where the programming language exposes details of how the hardware stores and executes software. Building from the design concepts covered in CS 2102, this course covers manual memory management, pointers, the machine stack, and input/ output mechanisms. The course will involve large-scale programming exercises and will be designed to help students confront issues of safe programming with system-level constructs. The course will cover several tools that assist programmers in these tasks. Students will be expected to design, implement, and debug programs in C++ and C.

Intended audience: computer science and computer engineering students with substantial prior object-oriented programming experience.

Recommended background: CS 2102.

Recommended background: CS 2303 or CS 2301, and CS 2011.

CS 3013: Operating Systems

Cat. I

This course provides the student with an understanding of the basic components of a general-purpose operating system.

Topics include processes, process management, synchronization, input/output devices and their programming, interrupts, memory management, resource allocation, and an introduction to file systems.

Students will be expected to design and implement a large piece of system software.

Intended audience: computer science majors and others interested in studying the software and hardware components of computer systems.

Undergraduate credit may not be earned both for this course and for CS 502.

CS 4241. WebWare: Computational Technology for Network Information Systems.

Cat. I

This course explores the computational aspects of network information systems as embodied by the World Wide Web (WWW). Topics include: languages for document design, programming languages for executable content, scripting languages, design of WWW based human/computer interfaces, client/server network architecture models, high level network protocols (e.g., http), WWW network resource discovery and network security issues.

Students in this course will be expected to complete a substantial software project (e.g., Java based user interface, HTML/CGI based information system, WWW search mechanisms, etc.).

Recommended background: CS 2102 and CS 3013.

CS 4513: Distributed Computing Systems

Cat. I

This course extends the study of the design and implementation of operating systems begun in CS 3013 to distributed and advanced computer systems.

Topics include principles and theories of resource allocation, file systems, protection schemes, and performance evaluation as they relate to distributed and advanced computer systems.

Students may be expected to design and implement programs that emphasize the concepts of file systems and distributed computing systems using current tools and languages.

Intended audience: computer science and computer engineering majors.

Undergraduate credit may not be earned both for this course and for CS 502.

Recommended background: CS 3013 and a knowledge of probability, such as provided by MA 2621.

CS 4536. Programming Languages.

Cat. II

This course covers the design and implementation of programming languages. Topics include data structures for representing programming languages, implementing control structures (such as functions, recursion, and exceptions), garbage collection, and type systems. Students will be expected to implement several small languages using a functional programming language.

Intended audience: CS majors and minors interested in understanding how programming languages work and how to implement their own small languages.

Recommended background: CS 2303, CS 3133, and experience programming in a functional language (as provided by CS 1101 or CS 1102).

Undergraduate credit may not be earned for both this course and CS 536.

This course will be offered in 2011-12 and in alternating years thereafter.

CS 502: Operating Systems

The design and theory of multiprogrammed operating systems, concurrent processes, process communication, input/output supervisors, memory management, resource allocation and scheduling are studied. (Prerequisites: knowledge of computer organization and elementary data structures, and a strong programming background.)

Appendix F

Box Plots

Box plots, also known as box and whisker plots, are simple graphs that display a summary of a group of data. Specifically, they display the minimum (min), maximum (max), and median values of a data set, along with the first (Q1) and third (Q3) quartiles. These data points are placed along the axis of a graph. A box is drawn that has sides that intersect the Q1 and Q3 values. A line is drawn through the box such that it intersects with the median value. A line, or whisker, is then drawn from each end of the box to the max and min values.

A sample box plot can be seen in Figure F.1. These box plots are drawn on the vertical axis. The plot for Series A has a min of 2, a max of 7, and a median of 4.5. The Q1 of the data set is 3.5 and the Q3 is 6. Given these data points, a box is drawn with sides intersecting

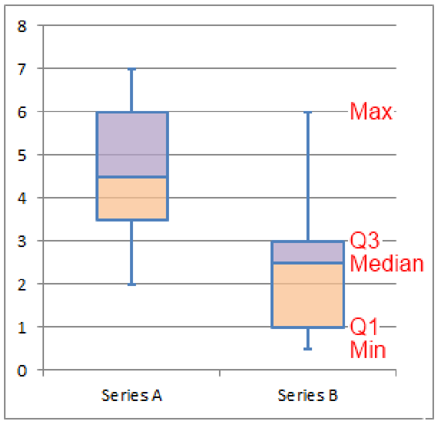
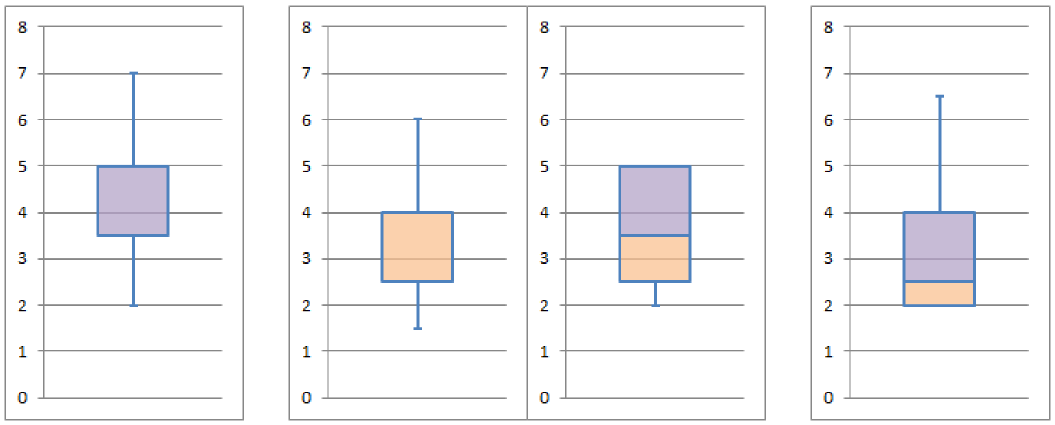


Figure F.1: Normal box plots



(a) Median = Q1 (b) Median = Q3 (c) Max = Q3 (d) Min = Q1

Figure F.2: Squished box plots

3.5 (Q1) and 6 (Q3), with a line intersecting 4.5 (median). Whiskers are then drawn from the box down to 2 (min) and up to 7 (max).

Sometimes, when there is a small range of data, you can run into situations when two or more of the values used to make a box plot are the same. To help with these situations, the box is two-colored, with purple always used to color the part of the box between the median and Q3, and orange always used to color the part of the box between Q1 and the median. Examples of this are shown in Figures F.2b and F.2a. In cases where the max is the same as Q3 or the min is the same as Q1, the whisker is simply not displayed. Examples of this are shown in Figures F.2c and F.2d.

Bibliography

[1] *About OpenMP and OpenMP.org* (2011). URL http://openmp.org/wp/about-openmp/. [Online; accessed 22-March-2011].

[2] *Effective Go* (2011). URL http://golang.org/doc/effective_go.html#concurrency. [Online; accessed 22-March-2011].

[3] *Getting Started with Erlang User's Guide - Concurrent Programming* (2011). URL http://www.erlang.org/doc/getting_started/conc_prog.html. [Online; accessed 22-March-2011].

[4] *History of Erlang* (2011). URL http://www.erlang.org/course/history.html. [Online; accessed 22-March-2011].

[5] *Implicit Parallelism in Fortress* (2011). URL http://projectfortress.sun.com/Projects/Community/wiki/ImplicitParallelismInFortress. [Online; accessed 22-March-2011].

[6] *Java Message Service (JMS)* (2011). URL http://www.oracle.com/technetwork/java/index-jsp-142945.html. [Online; accessed 22-March-2011].

[7] Biancuzzi, F. and Warden, S. *Masterminds of Programming*. O'Reilly Media, Inc, Sebastopol, CA (2009). ISBN 978-0-596-51517-1.

[8] Brookes, S.D., Hoare, C.a.R., and Roscoe, a.W. *A Theory of Communicating Sequential Processes*. *Journal of the ACM*, 31(3):pp. 560--599 (June 1984). ISSN 00045411. doi:10.1145/828.833. URL http://portal.acm.org/citation.cfm?doid=828.833.

[9] Haller, P. *Scala Actors: A Short Tutorial* (2008). URL http://www.scala-lang.org/node/242. [Online; accessed 22-March-2011].

[10] Hanselman, S. *Twitter / @Scott Hanselman* (2011). URL http://twitter.com/shanselman/status/51723484350517249. [Online; accessed 28-March-2011].

[11] Hewitt, C., Bishop, P., and Steiger, R. *A universal modular ACTOR formalism for artificial intelligence*. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artifical Intelligence*, pp. 235--245. Morgan Kaufmann Publishers Inc., Stanford, USA (1973).

[12] Hoare, C.a.R. *Communicating sequential processes*. *Communications of the ACM*, 21(8):pp. 666--677 (August 1978). ISSN 00010782. doi:10.1145/359576.359585. URL `http://portal.acm.org/citation.cfm?doid=359576.359585`.

[13] Lee, E.A. *The Problem with Threads*. *Computer*, 39(5):pp. 33--42 (May 2006). ISSN 0018-9162. doi:10.1109/MC.2006.180. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1631937`.

[14] Pankratius, V. and Tichy, W.F. *Parallelizing BZip2 A Case Study in Multicore Software Engineering*. *Source*, (December) (2008).

[15] Pike, R.C. *Advanced Topics in Programming Languages: Concurrency/message passing Newsqueak* (2007). URL `http://video.google.com/videoplay?docid=810232012617965344#`.

[16] Seibel, P. *Coders at Work*. Apress, New York (2009). ISBN 978-1-4302-1948-4.

[17] Silberschatz, A., Gagne, G., and Galvin, P.B. *Operating System Concepts*. John Wiley & Sons Inc., Hoboken, NJ, 7th edition (2005). ISBN 0-471-69466-5.

[18] Sutter, H. and Larus, J. *Software and the concurrency revolution*. *ACM Queue*, 3(7) (2005).

[19] Wikipedia. *Sieve of Eratosthenes --- Wikipedia, The Free Encyclopedia* (2011). URL `http://en.wikipedia.org/w/index.php?title=Sieve_of_Eratosthenes&oldid=418089133`. [Online; accessed 18-March-2011].