CS452 – S12

Final Examination

Length of Examination. 24 hours.

Time of Examination. 09.00, 7 August to 12.00, 9 August, 2012. Students may start the examination at any time during this interval, and must end writing at the earlier of twenty-four (24) hours after starting and the ending time.

What to Do. Read the examination with care from front to back. You are required to do three questions: questions 1 & 2, plus one question chosen from questions 3, 4 & 5.

Allowable Assistance. Students who wish to get a good mark on the exam are expected to do some research while writing it. All material that is not original with the student, or that comes from course materials, must be properly referenced.

Questions During the Examination. Students are expected to send e-mail to the instructor when they start reading the examination, which is their 'starting time'. Questions e-mailed to the instructor will be answered on the newsgroup to make them equally available to all students. Six (6) hours after the first student starts, questions will be closed, so that all students have a stable exam while they are writing.

Handing in the Examination. Examinations should be written on a computer, the results put into a PDF document, and the PDF e-mailed to the course account (cs452) in the student environment. The time of the e-mail is considered to be the time at which the student ended the exam.

Marking. Please see the section in the course introduction describing how the examinations are marked. Questions on this subject are answered at a length that will surely put you to sleep, to the detriment of exam writing. In effect, it amounts to the following.

1. You can get at most half of the marks available by giving correct answers that fail to go beyond what is explicitly asked.
2. You can get up to three quarters of the marks by introducing examples from outside the course and/or reference material.
3. To get full marks you must introduce ideas of your own.

.Question 1. Mutual Exclusion.

It is a truism hardly requiring proof that synchronization primitives must be able to emulate one another, albeit with different performance. One synchronization primitive that you have encountered from time to time is mutual exclusion, in which code that manipulates a shared resource can be executed by only one thread or process at a time.

Part a. Lock Servers. A lock server is an easy way to implement mutual exclusion, and is very much in the style of the cs452 use of message passing. Write in a rough way the code of a lock server and of two clients who are sharing a resource. Explain how and why it works, giving the strong and weak points.

This style of implementing mutual exclusion is often called 'self service'. It's like pumping your own gasoline: the guy in the little booth turns on the pump for you, and once it's on you do the work. Then, there are all sorts of bad things you can do, such as filling your tank and driving away. An elaborate infrastructure of banks, credit card companies, police and courts do the work.

Part b. Performance of a Lock Server. There are two performance issues associated with a lock server: overhead and enforcement. Using the performance measurements you made of your kernel, estimate the amount of overhead associated with gaining and releasing a lock. Roughly how much code would the client have to execute to have the overhead a small fraction of the client's load?

Enforcement is more tricky. There is code that the client wants to execute. Does the design you described in Part a enforce mutual exclusion? Describe at least two problematic cases, and how you would handle them.

Part c. Proprietor. The proprietor offers synchronized access to a resource in a full service model, like a plumber. The plumber does not bring plumbing tools for you to use, but uses them on your behalf. The style of server used in the course does not give you permission to execute code but does the execution for you. What are its advantages and weaknesses compared to a lock server? How does its performance compare to the performance of a lock server? Be as quantitative as you can. How does this solve the enforcement problem? Or does it?

Part d. Executing Code Supplied by the Client. When the plumber comes to my house I like to watch. I learn some things. I also see things being done counter to what I have learned previously. But I have discovered that giving the plumber advice about how to do the work is a bad idea: offering suggestions slows down the service and irritates the plumber. (Snatching his tools and using them to do the job is even worse!) Could we design a proprietor that would do better?

Here's the idea. The client provides the proprietor with code, possibly a function pointer, that manipulates the resource as the client wishes, while the proprietor keeps competing clients in its sendQ.* Discuss this idea, focussing on the balance between strengths and weaknesses

Part e. The Plan9 Plumber. Talking about plumbers, Plan9 generalizes the Unix pipe into a component called the plumber, which copies its input to its output, converting its format if necessary. Think of it as a collection of codecs. The plumber requires mutual exclusion because many codecs use special purpose hardware. Plan9 uses CSP (Question 5) for communication. Compare two of mutual exclusion, Send/Receive/Reply and CSP as methods for implementing the plumber.

---

* This sounds like a crazy idea, but it's been done. In the 1980s X had a competitor, NeWS. Jim Gosling programmed it in PostScript (!), and the application developer could supply PostScript that, run by the server, manipulated shared resources. Gosling's second attempt to do the same thing was Java.

Question 2. Memcpy.

When you were first exposed to C, I expect that you marvelled, as I did, at this code.

```
void strcpy( char *s1, char *s2) {
     while ( *s1++ = *s2++ ) ;
}
```

It copies a string to a different location in memory, byte by byte. This is the most basic method for copying a block of memory from one location to another. There are at least a million things wrong with it: it assumes that the last byte is the first occurrence of null after the start of `s2`; no bounds checking is performed; nor is any checking of buffer sizes done. Yet, in an environment like your kernel, where fast execution is essential, and you are well isolated from malicious or incompetent programmers, it might be a first implementation of block copying.

Part a. Compilation. Submit this scrap of code to the compiler used in the course (not optimized), and take a look at the code it emits. Explain why each part of it was emitted by the compiler, indicating whether you consider it to be necessary or not.

Part b. Efficiency. Re-write the compiler's effort, getting the performance as good as possible while copying byte by byte. Use the following definition of performance.

1. There are two types of instructions: one accesses memory; the other does not. Let the first take mem cycles, the second cpu cycles.
2. Some instructions are executed only once; others are executed once per byte. Multiply the second by n, the number of bytes copied.
3. Give the performance in terms of mem, cpu and n.

Calculate the performance of the compiler's code and of your own. Explain in general terms why the results are different.

Part c. Words. If you are willing to align your memory so that copies always start and end on four byte boundaries you can use the following slightly altered code

```
void strcpy( int *s1, int *s2) {
     while ( *s1++ = *s2++ ) ;
}
```

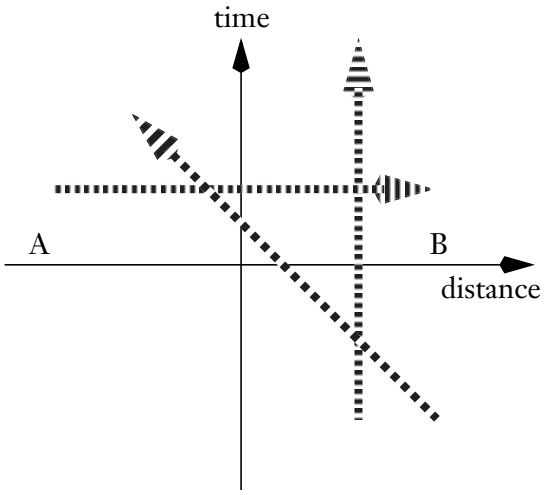
What additional assumption(s) must be made for this code to work? Compile the code with `-O2` optimization. Can you make code that is better? Calculate the performance of each and give a quantitative, but general, explanation of the value of hand optimization.

Part d. Even Better. Many of you did a further optimization using load & store multiple instructions. Assuming again that the data starts and ends on four byte boundaries provide assembly code that gets even better performance. Explain what special feature of the architecture makes the performance better, and by how much. (You may need to assume that you get the copy request with more information available. If so explain what it is and why you need it.)

Note. It's not part of the exam, but a possibly interesting titbit of knowledge. The earliest GPUs spent almost all their time doing `bitblit`, which is just hardware accelerated memcpy, and the result was an impressive performance improvement on 2D graphics. Improving the quality of hardware acceleration continues to be an important theme in GPU design. It seems interesting that continuing improvement in very humble functionality is such an important aspect of, for example, game and UI acceptability.

Question 3. Relativistic Reservations.

In the theory of relativity, particles in motion are drawn on space-time diagrams like the one to the right. The vertical arrow shows the space-time path of a particle at rest; the oblique line of a particle in motion; the horizontal line of a particle teleporting. When a train receives a route you can put the end points on the diagram at A and B. In reality, it would be hard to represent a reversing route like this, so we will break routes up into discrete segments separated by reverses. (A route with no reverses would be a single segment.



Part a. Travelling. On a space-time diagram show the path of a hypothetical train that starts from rest at A and travels with constant speed to B, where it remains at rest. In reality few trains can get up to full speed immediately. Draw a second path that better describes the space-time paths of the trains you used in your project. Explain why it differs from the first path.

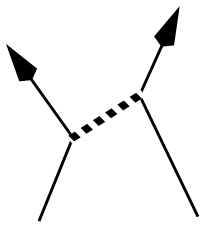
Part b. Collision Avoidance. Describe the system you implemented in your project – or were implementing, or would have implemented – to ensure that trains do not collide. Focus your explanation on a picture of it drawn on a space-time diagram. That is, at any moment your train is within a 'region of the space-time diagram that is forbidden to other trains. The forbidden region changes as the train moves. Put the realistic trajectory from Part a onto the diagram, marking on it points (not too many) where interaction with a server could cause the forbidden region to change. Associated with each point there is a space-time region that is forbidden: draw them on the diagram.

Each of the regions has a shape and size that is determined by assumptions you made when you decided how to design the system. Explain the shapes you drew in terms of the assumptions.

Part c. Doing Better. Improving a collision avoidance system so that you can run more trains on the track can now be described concisely and quantitatively: the forbidden regions must be made smaller. Can you make the forbidden regions smaller without violating your assumptions? If you can't, explain why; if you can explain how.

Big improvements require you to relax your assumptions. You had to assume that a train could stop any time. Assuming that trains don't stop unpredictably and follow their calibrated velocities how would the regions change shape. (Remember that calibrations are not perfect.) Draw a big space-time diagram of three trains, following each other, together with their reserved regions.

Part d. Feynman Diagrams. Richard Feynman influenced physics by inventing diagrams that are now widely used. The one to the right shows two negatively charged electrons repelling one another by exchanging a photon., which suggests that two trains might avoid colliding by repelling each other.



Notice that the photon takes time to travel from one electron to the other. The diagram resembles two engineers communicating to stay away from one another, and indeed it takes time for the message to travel, time during which the trains continue to travel towards one another. Using numbers you know from your project – train speeds, send/receive/reply times, stopping distances and others – work out whether or not you need to take this delay into account when designing a robust collision avoidance system.

Question 4. Double Deadlock.

In class we talked about deadlock, and how to break it when it occurs in the application domain. The main example we used was two trains travelling in different directions coming together on the same piece of track. If all works well they stop with one centimetre separating them, each waiting for the other to relinquish get out of the way.

Hint. Time line diagrams likely to make your answers in this question easier to understand.

Part a. Single deadlock. In the project demos we saw many different approaches to deadlock resolution, some complete, some partially complete, some only designed. Describe the strategy you implemented, or planned to implement. Describe how it resolves two trains stopped head on facing one another, taking into account that the code being run by each of the trains is exactly the same.

Part b. A Corner Case. One difficult to handle situation occurs when one train is trying to leave a siding while another train is attempting to enter the same siding, or another siding on the same tree of sidings. Describe in detail how your deadlock resolution method handles this case? (We saw this problem several times this term, after previous terms in which it occurred only rarely. Or possibly it occurred a lot and we just weren't smart enough to see it as a specific class of deadlocks.)

Part c. Double Deadlock. A more complex form of deadlock can occur, when one train is trapped between two trains that are heading toward one another. In class we discussed a single deadlock solution that used the ethernet collision avoidance algorithm, wait a random time before trying again, which works most of the time. Describe in detail an example of how it would work when trying to deal with a 'typical' double deadlock. (That is, ignore corner cases like the one described in Part b.)

Part d. N-way Deadlock. In computer science our solution to every problem is generalization*. Double deadlock, which is a generalization of deadlock, can be further generalized to N-way deadlock, where N-1 trains are trapped. Will the ethernet algorithm break deadlocks of arbitrary complexity? (As in Part c ignore corner cases, and don't worry about how long breaking the deadlock will take.) If you think so, sketch an argument that would convince me that you are correct; if not, give a counter example, explaining in detail how it doesn't work.

---

* Generalization and abstraction are nothing more than two sides of the same coin.

Question 5. Servers in Go.

As I mentioned in class the Go language is a project at Google, undertaken by former Plan9 people* who fled Lucent. Like Plan9 it uses CSP for synchronization and communication.

There are two common paradigms† of programming with multiple threads of computation and shared memory. One is the client/server model, which we discussed extensively in the course; the other is stream computation

Hint. If you are considering doing this question it would probably be a good idea to take a look at tutorial material on Goroutines at `golang.com`.

Part a. Client/Server Computation in Go. Write in Go, or pseudo-Go, a warehouse server task and two clients. The warehouse receives two types of request, one that provides a single item, the other that passes out packages, each of which contains three items. One client supplies items, one at a time; the other demands packages, also one at a time. Pay particular attention to the following.

1. Initialization. How do the tasks arrange to share the same channel?
2. Synchronization. How do you ensure that no task blocks indefinitely, and that the warehouse does not get overfilled?
3. Queuing. The warehouse should operate first-come, first-served.

Part b. Stream Computation in Send/Receive/Reply. There is a very ingenious example floating around the web, of stream computation in Go. It outputs the prime numbers, in order, using the sieve of Eratosthenes. The 'first user program' receives in a channel the natural numbers starting with 2. The first number it receives, 2, is its divisor. It then

1. outputs the divisor,
2. creates a channel,
3. instantiates a copy of itself that receives on the channel, and
4. forever passes to the channel numbers it receives that are not divisible by its divisor.

Draw a 'task' diagram of this organization to convince yourself that it actually does what it claims. Then provide an implementation of the single task that creates the sequence using Send/Receive/Reply, omitting unnecessary components.

Part c. Natural Computation. The design you created in Part b probably did not use what you have been taught to be natural client/server task organization, even though each individual task has a familiar organization,

1. initialization, in which communication is set up, followed by
2. an infinite loop where the actual work gets done.

Similarly, the design in Part a seems, at least to me, a contrived and dangerous‡ use of CSP. In your education so far you have seen a variety of synchronization and communication primitives. Describe the natural use of each, concentrating on the nature of the problem it solves most naturally. Can you generalize the problems into a single problem? Or is computer science, unlike other sciences, naturally heterogeneous?

---

* Ken Thompson, Rob Pike, Russ Cox.
† Dread word.
‡ What if the client decides never to receive?