University of Waterloo

cs452 – Final Examination

Fall 2009

Student Name:	_____
Student ID Number:	_____
Unix Userid:	_____

Course Abbreviation:	cs452
Course Title:	Real-time Programming

Time and Date of Examination:	12.30, December 10, 2009 to 15.00, December 11, 2009.
Duration of Examination:	26.5 hours.
Number of Pages:	7.

Rules of the Examination.

1. The cover page does not make sense. It is required by the registrar's office. Marks appear in all questions to give you an idea of the amount I expect to see for each part of each question. They may be slightly altered during marking.
2. You must do questions 1–4 and two (2) of questions 5–8.
3. You must work independently.
4. You may use any source of information you want on this examination. Information from sources you consult MUST be referenced. (Your memory, course notes and lectures are the only exceptions.)
5. Quite often you lose marks because what you are trying to say is not easy for me to figure out. Diagrams and examples reduce ambiguity and increase my understanding.
6. I require answers to be in PDF format (whatever.pdf). One to two pages (200 to 400 words, or less if there are diagrams) is about right for each question. Put your name, student number and userid on every page.
7. Your answers may be submitted either by bringing a printed copy to my office (DC2111) or by e-mailing them to me at `wmcowan@cgl.uwaterloo.ca`.
8. A strategy that works well for me is to read the exam twice, then do something else for a couple of hours, then plan my answers, rest again, and finish by writing them. The total writing time should be about three hours.
9. You will gain marks for the thoughts that you contribute to your answers. Write other people's thoughts only to the extent that I need them to understand your answer.
10. When the examination says 'your kernel' it means the kernel you actually created, not an ideal kernel or the kernel you wish you had created. When the examination says 'your OS' it means your kernel plus the other tasks (couriers, notifiers, servers) on top of which applications run. When the examination says 'your train application' it means the application you tried to create in what you would consider to be its final form.
11. Read each question carefully, and more than once. More marks are lost because of misunderstood questions than from any other single cause.* To show that you read this far, for one mark put the phrase 'Sargon the Magnificent' at the top of your first page. The advice to read at least twice is self referential.

---

* There is an interesting pathology I see in a few students. They expect questions on exams to be difficult or tricky. If a question seems straightforward and easy, they look for another interpretation, and continue doing so until they find a difficult and tricky one. Such students, needless to say, do badly on exams regardless of how well they know the material.
I do my best to make the exam easy to understand, and I do not ask questions with tricks intentionally.

DO ALL OF QUESTIONS 1–4.

Question 1. Task Creation in Your Kernel.

Most of you, in your kernel and in applications that run using it, have a relatively static task structure. When an activity starts tasks are created and initialized, after which they interact in a fixed pattern as long as the activity persists.* This makes sense: static structures are easier to create, debug and maintain. Most of the time, they are also more efficient.

- **(i)** (6 marks) For your kernel estimate on a logarithmic scale† the time, in machine instructions, required to create and initialize a worker task, breaking the process of creation into discrete parts and estimating the time for each part.
- **(ii)** (3 marks) Similarly estimate the time required to re-activate a pre-created worker task.
- **(iii)** (5 marks) Describe a situation in which it would be advantageous to create worker tasks, even though you have no way to destroy them. How should an application make the choice between waiting for an existing worker to come free and creating a new worker?

Question 2. Message Passing in Your Kernel.

In applications using your kernel a task should not Reply to itself. Suppose a task tries to do so, which it can because the line of code

```
error = Reply( MyPid( ), &reply, sizeof( ReplyType ) );
```

will not be rejected by the compiler.

- **(i)** (1 mark) What error does your kernel return?
- **(ii)** (2 marks) What is tested inside your kernel that produces this error?
- **(iii)** (6 marks) List in detail the actions that occur between the execution of `Reply` in user code and the error return.
- **(iv)** (3 marks) How does your train project respond to this error? Why?
- **(v)** (4 marks) How *should* user code respond to this error? Explain your answer.

Question 3. Hardware Interrupts.

In your kernel you used the ordinary hardware interrupt to provide prioritized service to hardware devices. The ARM processor also supports what it calls a fast interrupt (FIQ mode), which is higher priority than the ordinary hardware interrupt.

- **(i)** (3 marks) Read the ARM processor documentation about the fast interrupt, and describe how it differs from the ordinary one.
- **(ii)** (6 marks) On the basis of what you have read describe how you think the ARM designer expects that a kernel will use the fast interrupt. Give details and examples.
- **(iii)** (3 marks) Is there any sensible use of the fast interrupt for improving the performance of your kernel? If 'yes', explain what it is; if 'no', explain why it is not useful.

Question 4. Calibration.

The train set has three problematic features that usually exist when programs control real-time real-world systems.

1. Feedback about the state of the system is only intermittently available.
2. Feedback about the state of the system is not instantaneous.

---

* Your train applications, which implement a single system with its overall goals defined at compile time, consist of a collection of activities that persist until the end of execution.

† On the logarithmic scale I desire, the only values are 1, 3, 10, 30, 100, 300, 1000, ...

3. Response of the system to changes in control parameters is slow.

Calibration of the trains is the solution to such problems: a good calibration makes it possible to predict where each train is now and where it will be in the future.

There are three aspects of train kinematics you may have measured:

1. the stopping distance for each speed value,
2. the velocity of the train for each speed value, and
3. the deceleration and acceleration profiles.

In fact, you probably measured them in that order. For example, if you measured only one of them it was probably the stopping distance.

(i) (6 marks) Give an example of each problematic feature from your train application and explain why it makes train control difficult.

(ii) (3 marks) Explain the meaning of the oxymoronic phrase 'predict where each train is now', and why it is important.

(iii) (7 marks) What train control capability do you have when you know only the stopping distance? What extra capabilities do you acquire when you measure the train velocity? What extra capabilities do you acquire when you measure the profiles? In your answer include how you use each capability in your project.

Do any two (2) of Questions 5–8.

Question 5. Response Time.

For your train project response time is an important performance parameter of your kernel. If the response time to a sensor is too slow, for example, you may not switch the turn-out soon enough, and the locomotive, on top of the turn-out when it switches, derails. You need a method to get rough estimates of response time.

The key concept is the *critical instant* of an event, which occurs when the event and all events of higher priority occur at the same time. Thus, the worst case response time for the event must take into account both the processing time for the event itself plus the processing time for all higher priority activities. (It is even possible that some of these activities must be processed more than once, if they re-occur often enough. ('Higher priority' should be understood generally: tasks in the send queue of a server, for example, will occur before a new request is handled, even if the tasks themselves have a lower priority.)

To be concrete let's try to estimate the worst case response time from a timer interrupt until the moment when the clock is incremented inside the clock server.

- **(i)** (6 marks) List (down to the granularity of context switches) in order the processing steps that must occur between a timer interrupt and the corresponding clock update. For each item in the list provide a logarithmic estimate* of the processing time (measured in machine instructions).
- **(ii)** (6 marks) List all the higher priority processing that would be done in the worst case, and provide estimates of processing time for each.
- **(iii)** (2 marks) What is the fastest clock period you would be willing to guarantee?
- **(iv)** (4 marks) If it were only necessary that the clock be eventually correct what is the fastest clock period you could support? Explain your answer. (Hint. This depends on the average response time, not the worst case one.)

Question 6. Segmented Stacks.

Initial implementations of kernels usually give every task a stack of the same size, which is a simple and robust place to start. But when memory is short this approach is not good enough: a courier, for example, requires much less stack than a route finder. Unfortunately, except in the simplest of cases it is impossible to bound a stack tightly using code analysis, and if you used different stack sizes† in your kernel you most likely chose them by intuition, accepting the weird bugs that happen when one stack overflows into another. Stacks that dynamically increase their sizes would make this design problem a lot easier.

An ancient solution to this problem‡, revived in Go, is the *segmented stack*. Each time the stack is incremented, code tests to see if the increment will overflow the current segment. If so, it gets an unused segment and puts the increment there, suitably updating the stack pointer. Each time the stack is decremented, code tests to see if the decrement leaves the current segment empty. If so, it gives back the empty segment and updates the stack pointer accordingly.

Clearly, the less often the stack is incremented or decremented the better segmented stacks work. Thus, it pays off to gather together code that increments or decrements the stack. The result is a few chunks of stack manipulation code. To be concrete, 'stack manipulation' moves the stack pointer in order to allocate or de-allocate local storage. We want to do this as little as possible.

- **(i)** (6 marks) Consider an application running on top of your kernel. Much stack manipulation is done while the kernel is initializing itself. After initialization is complete, there remain two

---

* On the logarithmic scale I desire, the only values are 1, 3, 10, 30, 100, 300, 1000, ...

† Choosing to put big tables in static blocks of memory presents the same estimation problems as determining stack size.

‡ PL/I implementations used segmented stacks in the 1960s.

occasions when stack manipulation is done inside the kernel. A different type of stack manipulation is scattered throughout kernel and user code. Identify the three occasions, and tell why the stack needs to be manipulated in each case.

(ii) (6 marks) As a segmented stack is incremented linkage data must be retained to be used when the stack is later decremented. Possible storage locations for this data include

1. the kernel, possibly in the task's task descriptor,
2. the task, possibly on its stack, or
3. other memory, like the MMU tables, with a special register pointing to it.

Give one good feature and one bad feature of each. Features for different locations should be distinct.

(iii) (2 marks) For one occasion the compiler can add stack segmentation code. Describe how the code would handle segmentation. (Assume that the segmentation data is maintained on the user's stack.)

(iv) (4 marks) For the two occasions explicit kernel programming is required. Describe how the code added to the kernel would operate. (Assume that the segmentation data is maintained on the user's stack.)

Question 7. Open Hardware

Open hardware architectures present kernel developers with hard problems. For example, compare MS Windows with MacOS. Maintaining compatibility, consistency and even backward compatibility is easy for MacOS, which runs on a closed collection of hardware architectures controlled by the same firm that develops the OS. Windows, on the other hand, must cope with any available hardware. We benefit from Windows, even if we don't use it, because it encourages hardware innovation. On the other hand, Windows drives us crazy because its kernel, which includes a humungous collection of device drivers, is huge, and because providing bug-free and consistent device drivers for such a disparate collection of hardware is almost impossible.*

One motivation of the design of the kernel you wrote was to be as portable as possible. Suppose you have the Notifier manage the hardware. Then, in theory all a new device requires is a new Notifier.

(i) (8 marks) What is the minimal set of actions that the Notifier must perform for each of the following devices:

1. a digital to analog converter doing output from a 100 byte buffer,
2. an analog to digital converter delivering one 16-bit input word at a rate of 44KHz,
3. a countdown timer, and
4. a 16-bit bidirectional parallel port.

(ii) (8 marks) Depending on the application the Notifier will use different tasks to interface its baseline functionality to input code. Suppose the devices above are used to support the following activities:

1. audio output from an iPod,
2. microphone input to a mobile telephone,
3. on-screen display of time, and
4. location sensing. (Location sensing is done by putting 16 mats on the floor, each attached to one bit of the parallel port. The bits are 0 when there is a force of more than 200 Newtons on the matt, and one otherwise.)

For each application describe what set of tasks you would use and give the function of each.

(iii) (2 marks) Compared to ethernet or USB all these devices and applications are very simple. Of course, ethernet and USB are designed to abstract devices at a different level. Give one advantage and one disadvantage of ethernet compared to the above examples with respect to making a kernel portable. Provide reasons for your answer.

---

* Regrettably, Linux suffers from much the same problem.

Question 8. Concurrent Sequential Processes (CSP).

CSP is an ancient* technique of communication and synchronization by message passing, which is making a comeback in Go. In CSP tasks communicate using a named channel. (In general channels can be two-way, but for the purposes of this question assume they are one-way only.) Naming channels makes it possible to provide either static or dynamic type-checking of messages.

A typical CSP language implementation has a channel type and two operators. In `occam 2` we have

```
CHAN OF BYTE keyboard
```

for a channel that accepts and provides individual bytes. A byte is written to the channel by

```
BYTE ch
ch := 'a'
keyboard ! ch
```

and read from the channel by

```
keyboard ? c
```

When the same channel is referenced in two separate tasks, data passes from one task to the other. A channel also provides synchronization because the sender blocks until the receiver reads each sent byte. (Go provides the same functionality with different syntax.

```
ch := 'a';
keyboard := make( chan byte );
keyboard <- ch;
```

and inside another thread of execution (goroutine)

```
c := <-keyboard;
```

has the same effect.)

Nested scope is an important property of most computer languages: a scope is a context in which specific names are known; nesting makes it possible to define scope unambiguously and to keep track of it using a stack. CSP allows multiple processes to exist within a single scope. Your kernel, in contrast, has nested scope within each task, but each task is independent of the others. The kernel provides globally unique task ids, used by tasks to communicate and synchronize, and a nameserver provides a global name space, mapping names to task ids.

- **(i)** (4 marks) Putting multiple processes within a single scope makes it possible for the compiler to type-check messages. Explain.
- **(ii)** (4 marks) In your kernel the Receive/Reply pair do not form a scope. Why is this essential for a well-functioning server?
- **(iii)** (4 marks) Provide CSP pseudocode showing how you can get the same functionality using channels. Explain why it works. (Hint. The value of a channel variable can be passed through a channel.)
- **(iv)** (3 marks) Tasks supervised by your kernel discover the task ids of other tasks by querying a name server, which replies a task id. Describe a situation in which it would be desirable to have a name server within CSP. Is it possible to provide a nameserver that is outside the scope of the processes that access it? Explain.
- **(v)** (3 marks) How could tasks that are separately compiled, and not linked, communicate using CSP?

---

* Once again from the 1960s. Do me a favour; don't read any books or journal papers with CSP in their titles. Most of them consist largely of what Gord Cormack calls 'chicken tracks'.