University of Waterloo

CS251 – Final Examination

Spring 2008

Student Name:	________________
Student ID Number:	________________
Unix Userid:	________________

Course Abbreviation:	**CS452**
Course Title:	**Real-time Programming**

Time and Date of Examination:	**13.00 to 15.30, August 8, 2008.**
Duration of Examination:	**150 minutes.**
Number of Pages:	**6.**
Additional Materials Allowed:	**None.**

To be filled in by marker:

Question	a	b	c	d	Total	Marker
1	/6	/6	/6	/6	/24	
2	/6	/6	/6	▨	/18	
3	/6	/6	/6	▨	/18	
4	/6	/6	/6	▨	/18	
Total					/60	

"This should only take about two hours..."

The questions in this examination do not have unique right answers. If you look for unique correct answers it will be hard and long.

The questions ask you to think and respond with your thoughts, based on what you have learned in the course. If you do this you should find it short and easy.

The questions are what some profs call discussion questions. Think and give reasoned opinions; integrate what you have learned this term, which is a lot.

READ THIS FIRST. The examination has four questions. Every student is required to do Question 1, which is actually a collection of short-answer questions. In addition every student is required to do any two (2) of Questions 2, 3 & 4. Read the questions carefully before choosing which two you will answer.

Then read them again. To get a good answer you should spend at least ten to fifteen minutes thinking and organizing your thoughts, then twenty to thirty minutes answering.

This is a short exam, I believe. I don't expect that any of you will stay until the end, so don't worry if you seem to be finishing it quickly.

Important. When the examination refers to 'your kernel' it means the kernel you actually created, not an ideal kernel or the kernel you wish you had created.

QUESTION 1 Short Answers.

1.a Interrupts.

(i) After an interrupt two actions must be done before interrupts are re-enabled. What are they?

(ii) One of these actions **must** be done in the kernel. Which is it? Why must it be done in the kernel?

(iii) One action may be done inside or outside the kernel. Where did you do it in your kernel? Explain in detail all the things you did in your kernel to prevent corruption by a second interrupt happening immediately after the one you are servicing.

1.b Couriers. A courier takes a message from a source to a destination, and you surely implemented them in your kernel.

(i) Describe the structure of courier code.

(ii) In some places couriers are necessary. Where? Why?

(iii) Did you use any optional (not necessary) couriers in your kernel or project? Why?

1.c Dynamic Memory Allocation. You were told not to use dynamic memory allocation in either your kernel or your project because it cannot be made real-time.

(i) The above statement is not quite true. What aspect of dynamic memory allocation is not real-time and how could you eliminate it? (Hint. Once you eliminate this feature dynamic memory allocation loses its point.)

(ii) The heap is usually put at the top of the data space and grows down, the stack at the top of data space and grows up. The **real** reason for not using dynamic memory allocation is what happens when stack and heap meet. What happens?

(iii) What would you have to add to your kernel to prevent stack and heap meeting? Describe how it would work.

1.d Task Structure. Task structure comes in four flavours

- static (Tasks are not created after interrupts are enabled for the first time.),
- almost static (Only a few tasks are created after interrupts are enabled for the first time.),
- almost dynamic (Most tasks are created after interrupts are enabled for the first time.), and
- dynamic (All tasks are created after interrupts are enabled for the first time.)

(i) One of the four is impossible for a kernel. Which one? Why?

(ii) Which category does your kernel fit into? Why did you make it that way?

(iii) Which category does your train project fit into? Why did you make it that way?

QUESTION 2 Cyclic Execution and Polling Loops

2.a The polling loop that you wrote at the very beginning of the course, and cyclic execution, which we discussed at the end of the course, are two different ways of doing almost the same thing. Briefly describe how each is structured, specifically indicating code that you could reuse if you were converting a polling loop to cyclic execution.

2.b When we think about polling loops we think about response time, the worst case amount of time that occurs between an interrupt and the execution of the first line of code responding to it. When we talk about cyclic execution we think about admission control, how to decide whether a new task submitted to the cyclic executive will be able to execute without overflowing the period of the cycle in the worst case.

(i) How do we calculate the worst case response time in a polling loop?
(ii) How do we calculate whether or not a new task can be admitted in cyclic execution?
(iii) I claim that admission control and worst case response time are very similar. Am I right? Give reasons for your answer.

2.c In your polling loop you learned two programming techniques for improving the response time of critical tasks:

1. putting a high priority task in the polling loop more than once, and
2. splitting a long-running low priority task into more than piece to open up a window for polling by other tasks.

What are the analogous techniques for cyclical execution?

QUESTION 3 Communicating Sequential Processes (CSP).

CSP is a interprocess (intertask) communication technique that has been implemented in some operating systems (e.g., Plan 9) and some programming languages (e.g., occam).

It uses named uni-directional channels. For example, the declaration

```
telegraph: CHAN
```

creates a channel called `telegraph`, on which two processes can communicate, one sending, the other receiving. Here is what that looks like in occam-like pseudo-code.

```
PAR
    reader( telegraph )
    writer( telegraph )
```

starts executing, in parallel, two processes (tasks) that are defined as follows.

```
PROC reader( receiver: CHAN )
    receiver ? request

PROC writer( sender: CHAN )
    sender ! request
```

One process sends a message, called `request`, to the other, which receives it. CSP synchronizes, just like your kernel. Whichever process arrives first at the channel blocks until the other arrives, when the buffer is transferred, after which they recommence executing independently.

3.a Your kernel is different in that it sends messages two ways in one transaction. What are the two messages? Which events happen at the same logical time when two communicating tasks synchronize by passing a message?

3.b CSP might try to imitate your kernel by using two channels, one for each message. Elaborate the code fragments above so that reader and writer send a message, first in one direction, then in the other. How is the resulting semantics different than that of Send/Receive/Reply?

3.c The server is a key concept of program structuring using your kernel. Why would it be hard to write a server using CSP? Could it be done using a channelserver, analogous to the nameserver in your kernel, which hands out channels in pairs?

QUESTION 4 This question is about your kernel, not about an ideal kernel.

Suppose you are doing your trains project with a partner who is either malicious or incompetent. (You choose.) Assume that the kernel cannot be reprogrammed by your partner, and that he can only cause havoc by writing tasks.

4.a Can your kernel be brought to a standstill by your partner?
If your answer is 'yes', give pseudocode of a task he could write that would make your kernel unable to continue supervising other tasks, and explain what happens to the task and your kernel as it runs.
If your answer is 'no', describe what you did to your kernel to protect it against malicious or incompetent application programmers.

4.b There are several ways that your partner, without incapacitating the kernel, can reduce almost to zero useful work your tasks are trying to do.
Describe two, and explain how they would work.

4.c The problems above occur because you and your partner are trying simultaneously to use a kernel that is designed on the assumption that it is running a single application consisting of cooperating tasks. Unix is designed on the opposite assumption: that many tasks are running, that they are independent, and that the operating system should keep them independent. Is there any reason for generalizing your kernel to keep apart non-cooperating tasks? If you think 'no', explain why. If you think 'yes' explain how it might be done.