University of Waterloo

cs452 – Final Examination

Fall 2008

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

Course Abbreviation:	**cs452**
Course Title:	**Real-time Programming**

Time and Date of Examination:	**10.00 December 8, 2008 to 23.59 December 9, 2008.**
Duration of Examination:	**roughly 150 minutes.**
Number of Pages:	**5.**

RULES OF THE EXAMINATION

1. You may use any source of information you want on this examination. Any information from sources you consult MUST be referenced. (Your memory, course notes and lectures are the only exceptions.)
2. I prefer answers in PDF format (whatever.pdf). If PDF is inconvenient then I accept plain text (whatever.txt) but you will have to stretch a little to make the diagrams that ar requested. One page (~500 words or less if there are diagrams) is about right for each question. Put your name, student number and userid on every page.
3. Your answers should be submitted using the submit command. I will use the time stamp of the submission to judge if your answer is on time, so allow a little time for the electrons to flow over the network.
4. 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 two hours at most.
5. Do question 1 and any two of questions 2–4.
6. You will be marked for the thoughts that you contribute to your answers. Write only enough of other people's thoughts that I can understand your answer.
7. 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.
8. 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 number three hundred and eighty-four at the top of your first page. The advice to read at least twice is self referential.

Question 1. Short Answers.

1.a Initialization. Immediately after your kernel starts executing it initializes itself, then creates the first task. Immediately after any task starts executing it initializes itself, then enters its `FOREVER` loop.

(**i**) Name three aspects of the system state that are initialized by your kernel.

(**ii**) Before a task starts executing part of its state is initialized by your kernel. Name three aspects initialized by your kernel.

(**iii**) Name two aspects of the state of the clock notifier that are initialized by the notifier after it begins executing.

1.b Detectives. A detective is a generalization of a notifier, discovering Boolean combinations of elementary events and notifying a client when they occur.

(**i**) Why must a detective be structured like a server?

(**ii**) A very simple detective could supply a time-out capability for a server that provides input from the train controller. Draw the required task structure and give pseudocode for the detective.

1.c Global Descriptor Table. One of the most annoying details of the x86 architecture is the global descriptor table (GDT).

(**i**) Describe two features the GDT can provide to an operating system and how they are provided.

(**ii**) The `sgdt` instruction sets the GDT register to a new value. Why does the CPU not use the new value when executing the instruction immediately following `sgdt`? In general terms, when does the CPU start using the new value?

1.d Task Synchronization. In your system inter-task synchronization is accomplished by the communication primitives: Send/Receive/Reply.

(**i**) Give an example in your OS where two tasks need to synchronize but, beyond that, have no need to exchange information.

(**ii**) Describe the sequence of events as they synchronize.

Question 2. Bottlenecks

2.a Cyclic Execution. When we talked in class about bottlenecks we did so in the context of cyclic execution, and I argued that worst cases in your system looked a lot like cyclic execution. A simple abstraction of the lowest level of execution of your kernel might include three external triggers (interrupts) that trigger changes in execution.

1. An interrupt from the timer. Suppose these happen at most once every 50 milliseconds.
2. An interrupt that occurs when a byte arrives on the UART. Suppose these happen at most once every 100 milliseconds.
3. An interrupt that occurs when the UART is ready to transmit a byte. Suppose these happen at most once every 100 milliseconds.

(This abstraction considers only one UART, and ignores both error and flow-control interrupts.)

(**i**) For each trigger draw a time line showing the sequence of task executions that occurs in your kernel when each type of trigger occurs. Use typical execution times for each task taken from your kernel to label the time lines.

2.b The Critical Instant. The critical instant occurs when all triggers appear at the same time. Answer the following questions assuming that all other tasks have lower priority than the lowest priority task in part **2.a**.

(**i**) For the critical instant draw a single time line indicating everything that occurs following it. Use the priorities you used in your OS.

(**ii**) What is the length of time from the critical instant until processing is complete for each trigger?

2.c Maximum frequencies. On the basis of the numbers calculated in part **2.b**(ii) you surely see that there is lots of spare time for your kernel to get through this bottleneck. Now, assume that the triggers speed up by a factor of k, trigger 1 occurring every $50/k$ mililiseconds, triggers 2 & 3 occurring every $100/k$ milliseconds.

(**i**) Define the criterion you would use for not getting through the bottleneck.

(**ii**) What is the maximum value of k your OS can handle, assuming it is doing nothing else. (Remember that triggers might occur several times before all processing is complete.)

(**iii**) Draw a timeline for your maximal k.

Question 3. Names

3.a Variables. Two tasks can have variables, including functions, with the same name.

(i) Why does this not lead to confusion?

3.b Tasks. Task ids (`Pid`s) must be unique, even including tasks that have died or been destroyed.

(i) Why?

(ii) Describe a scenario where the wrong thing would happen if you reused the `Pid` of a defunct task.

3.c Types. In class we said that named types are handy for message passing.

(i) Why are they handy?

(ii) Did you use type definitions for messages in your kernel? If so how did you implement them? If not describe why you thought they weren't worth the trouble, specifically mentioning formatting and parsing.

3.d NameServer. The NameServer is a server that can provide the `Pid` corresponding to the name under which a task registers itself.

(i) Why is it important to have a NameServer?

(ii) In your OS how did tasks discover the `Pid` of the NameServer?

(iii) Why does the NameServer require global coordination among the names under which tasks register themselves?

(iv) In a very big application global name coordination could easily become inconvenient or costly. As a solution you might define a hierarchy of namespaces like the ones in C structs or Java classes. Think about DNS service, and describe how a collection of NameServers could efficiently provide hierarchical namespaces.

(v) Draw a task diagram for your solution above, including couriers and other ancillary tasks.

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 or she can only cause havoc by writing user tasks.

4.a Stop the kernel. Is it possible for your partner to write a task with the following property. After the first time it is activated, the system continues to run but the kernel never executes another instruction?
If your answer is 'yes', describe how to do it.
If your answer is 'no', describe how your kernel protects against it.

4.b Stop the kernel. Is it possible for your partner to write a task with the following property. After the second time it is activated, the system continues to run but the kernel never executes another instruction?
If your answer is 'yes', describe how to do it.
If your answer is 'no', describe how your kernel protects against it.

4.c Crash the kernel. Is it possible for your partner to write a task that causes the kernel to crash the next time it is entered? (Assume that your partner does not know *a priori* where the kernel is loaded.)
If your answer is 'yes', describe how to do it.
If your answer is 'no', describe how your kernel protects against it.

4.d Appropriate the kernel. Is it possible for your partner to write a set of tasks programmed so that they are the only set of user tasks that ever run?.
If your answer is 'yes', describe how to do it.
If your answer is 'no', describe how your kernel protects against it.