University of Waterloo

cs452 – Final Examination

Spring 2010

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

Course Abbreviation:	cs452
Course Title:	Real-time Programming

Time and Date of Examination:	13.00, August 10, 2009 to 16.30, August 11, 2009.
Duration of Examination:	27.5 hours.
Number of Pages:	9.

Rules of the Examination.

1. You must work independently.
2. 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.)
3. 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 diagrams. Two pages (~800 words, or less if there are diagrams) is the longest answer you need for any question; some questions require must less. Put your name, student number and userid on every page.
4. Your answers should be submitted by e-mailing them to me at `wmcowan@cgl.uwaterloo.ca`.
5. 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.
6. 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.
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. 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.
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 phrase 'Sargon the Magnificent' at the top of your first page. The advice to read at least twice is self referential.

Do all of Questions 1–3.

Question 1. Scheduling in Your Kernel.

Earl has criticized the implementation of scheduling I give you in class. To refresh your memory here are the critical details.

1. There are N priorities.
2. For each priority there is a singly-linked list of task descriptors of ready tasks at that priority, with a head pointer for extraction and a tail pointer for insertion.
3. An N element array contains a pointer to a head/tail pair for each priority.
4. An integer n holds the array index of the highest priority non-empty priority.

1.a Insertion. Earl admits that insertion is constant-time in this implementation. Go through every operation done during insertion, showing that it is constant time.

1.b Extraction. Earl claims, however, that in this implementation extraction is not constant time, but is linear in N. I agree. List the operation done during extraction, saying whether the worst case of each one is constant-time or linear in N.

1.c Doing Better. Earl says, in my view correctly, that you can lessen the linear dependence to a logarithmic one by using a heap instead of an array. I say that

1. if N is small then a simple linear function outperforms a logarithmic one because its constant is smaller; and
2. if you really care about performance you are better to use a bit map*, plus the CLZ instruction†, for finding the highest priority non-empty queue in constant time for values of N up to 32.

Estimate the scheduling load of your kernel, and use your estimate to determine when each of the three alternatives – array, heap, bit map – is best, taking them as a function of number of priorities.

---

* The bit-map is a word in memory, managed by the scheduling software. If a priority queue is non-empty then the corresponding bit in the word is set.

† `CLZ rd,rm` sets register rd to the position of the highest set bit of rm in one clock cycle.

Question 2. Memory Allocation in Your Kernel.

You were told in class that you could not use dynamic memory allocation in your kernel because you cannot provide real-time performance guarantees for it.

2.a . The above statement is not precisely true and you did, in fact, allocate memory each time you filled in a task descriptor. You may have done this allocation during initialization of the kernel, but you can do it in constant time at run time. Describe everything that must be done to allocate memory for a task, and explain how it is constant time.

2.b . The problem is, of course, not allocation, but de-allocation followed by reuse, the cost of which depends strongly on allocation strategy. For example, describe why real-time memory re-use is infeasible if it is possible to allocate in any multiple of a byte.

2.c . Reasonable strategies for almost-constant-time allocation/de-allocation are similar to two of those discussed in Question 1,

1. keeping a free-list of constant-size blocks in a heap, and
2. using a bit-map of in-use blocks together with the CL*Z* instruction.
   - **(i)** Why is the linear strategy not feasible? Give numbers.
   - **(ii)** The bit map strategy can be made logarithmic in the number of allocatable blocks. Describe how and compare its performance to that of a heap.

Question 3. Context Switching

Context switching is the *sine qua non*[*] of operating system kernels.

3.a Software Interrupt Instruction.

(i) Why is there a software interrupt instruction when it is possible to achieve the same result by writing to the ICU, which can trigger a hardware interrupt?

(ii) Why is it unnecessary to save IP on a software interrupt?

3.b The Link Register.

(i) In the ARM architecture the link register, r14, plays a special role. What is it?

(ii) When a hardware interrupt occurs two different link register values exist. What are they and how is each one used?

(iii) In your kernel where did you save each of these values?

(iv) You don't have to worry about the existence of two values with software interrupts. Why?

3.c Fast Interrupts. Fast interrupts can improve efficiency by removing the necessity of saving state when a hardware interrupt occurs.

(i) Explain how you can do useful computation without saving state.

(ii) The Unix strategy of communicating between fast interrupt handlers and the state works like this.

- Each different fast interrupt source has a bit in a bit mask.
- When the interrupt occurs the handler saves volatile data in a kernel-accessible location, and sets its bit.
- When a context switch next occurs the kernel checks the bit mask and does whatever needs to be done to continue processing the interrupt-supplied data.

What is unsatisfactory about this strategy for your kernel?

(iii) What would happen if you expected the software interrupt instruction within the fast interrupt handler?

(iv) Why did I ask the previous question? (*Hint*. Suppose you wanted to continue processing fast interrupt data only occasionally.)

---

* Without which nothing.

Do any two of Questions 4–9.

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

4.a . Give an example of each problematic feature from your train application and explain why it makes train control difficult.

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 speed values you planned to use,
2. the steady-state velocity of the train for each speed value you planned to use, and
3. the deceleration and acceleration profiles from one speed value to another. (A profile tells you the time-dependence

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

4.b . Suppose you know only the stopping distance. What is possible to know about the train position when the train is stopped, and how should your software stop the train in order to know it?

4.c . When you know the stopping distance and the steady state velocity of the speed you are using it is possible to stop the train at a pre-specified location most of the time? Describe how to do so. When is it not possible to predict precisely where the train will stop?

4.d . Suppose now that you have calibrated all three aspects of train kinematics. Describe how you can stop at a precise location in the case where it is not possible to do so without the deceleration and acceleration profiles.

Question 5. 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, Go).

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

```
telegraph: CHAN OF <type>
```

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 OF <type> )
    request: <type>
    receiver ? request

PROC writer( sender: CHAN OF <type> )
    request: <type>
    sender ! request
```

`writer` sends a message, called `request`, to `reader`, which receives it. `<type>`, in the declaration of `telegraph`, specifies the type of data that can be written on the channel.

5.a . Notice that the structure in the sample is very static, and that static type-checking is possible. Explain what parts of the code the compiler must consider in order to check type consistency. Is the qualification of `CHAN` by `<type>` necessary for static type checking in this example? Is your answer true for all examples?

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.

5.b 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?

5.c 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 does the resulting semantics differ from Send/Receive/Reply?

5.d The server is a key concept of program structuring for applications using your kernel. Describe how you could write a server using CSP, and how it would be accessed by a client. (Rough pseudocode with explanations is probably the easiest answer to write.)

Question 6. Cyclic Execution and Polling Loops

6.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.

6.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.

6.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?

6.d . Cyclic execution is very natural for a device like a mobile telephone.

1. At a level below the operating system, time is divided into basic periods.
2. Time-critical tasks, like animation, playing music, or communicating with the line card in the ground station are scheduled at the beginning of each period. They are subjected to admission control when they start to ensure that adequate time is available for them.
3. The leftover time in the period is provided to an operating system, possibly your kernel, which provides asynchronous, non-time-critical interaction with the user. The OS is told each time it is scheduled the amount of time it has to run, and is expected to yield the processor in time for the next cycle to start.
4. The OS can insert new time-critical tasks into the period, when the user wants to start a new activity, and delete old tasks from the period when an activity is complete.
   - **(i)** Obviously, such an architecture is interesting only when the OS can interact with the time-critical tasks. Describe possible who must block on whom, and why, when this interaction occurs.
   - **(ii)** The communication is naturally CSP-like, which provided a different way of structuring a server. Describe how you cold have time-critical code supply services to asynchronous tasks.

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. T the same time, Windows drives us crazy because it is huge, because it must include a humungous collection of device drivers, and is also buggy, because providing bug-free 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 Notifiers managing the hardware. Then, in theory, all a new device requires is a new Notifier.

(i) What are the minimal actions that the Notifier must perform for each of the following devices:
1. a digital to analogue converter doing output from a 100 byte buffer,
2. an analogue 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) 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) Unfortunately, even if the Notifier completely encapsulates the hardware there are consequences for the kernel because it must know about the range of possible interrupts. Furthermore, tasks like warehouses, couriers and servers will inevitably have hardware-specific types. Suggest at least two global strategies for minimizing such problems.

(iv) Compared to ethernet or USB every one of these devices and applications is very simple. Of course these two protocols 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.