University of Waterloo

cs452 – Final Examination

Spring 2015

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

Course Abbreviation:	cs452
Course Title:	Real-time Programming

Time and Date of Examination:	20.30, Tuesday, 4 August, 2015 or 09.30, Wednesday, 5 August, 2015 or 20.30, Wednesday, 5 August, 2015.
Duration of Examination:	26.5 hours.
Number of Pages:	7.

Rules of the Examination.

1. Do question 1 and two of questions
2. You must work independently. Phoning your partner to find out what is in your kernel is not considered independent.
3. 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.)
4. I prefer answers in PDF format (whatever.pdf). Three pages (~1000 words, or less if there are diagrams) is as long an answer as you need for any question, but only if you write the appropriate three pages. Regurgitating the question or course notes get few or no marks. (See part 7., below.) Put your name, student number and userid on every page.
5. Your answers should be submitted by e-mailing them to me at `wmcowan@cgl.uwaterloo.ca`.
6. A strategy that worked well for me as a student was 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.
7. Please remember that the questions are open-ended: you get most of your marks from going beyond simple answers; explicit instructions are intended as prompts to get you started in the right direction. Answering only what they request gets you about half the available marks for a question. The other marks come from ideas that go beyond the question as asked.*
   You gain marks for the thoughts that you contribute to your answers. Write other people's thoughts, including mine, only to the extent that I need them to understand your answer.
8. 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/project' it means the application you planned to create, in what you would consider to be its final form.
9. All measurements and estimates must have units. If your unit is ticks translate it into milliseconds using the size of tick in your kernel.
10. There may be places in the examination where you want to make assumptions. Do so, being certain that you explain your assumptions and how they are related to what you are saying.
11. Read each question carefully, and more than once. More marks are lost because of misunderstood questions than from any other single cause.
12. In all questions you should give your reasoning. More marks are given for reasoning than for correctness.
13. The cover page exists only to fulfil the registrar's regulations.
14. Read part 7.,above, once more.

---

* To understand why I handle the exam as described please consult the Introduction to the course.

Do Question 1.

Question 1. User State in the Kernel.

You are told in class by the instructor and in the lab by TAs to keep the minimum possible task state in the kernel. This question examines ways in which task state might be kept in the kernel, and problems it might cause.

When this question asks you to draw things draw them schematically, showing only what you think is important.

1.a. In the task descriptor. One method for storing task state is to leave room for it in the task descriptor. In other words, when the task descriptors are declared during initialization, each is large enough to hold the entire state of a task. This is common in many embedded applications and recommended in many engineering OS classes.*

- **(i)** How big, in 32-bit machine words, would the task descriptor be?
- **(ii)** Draw what's in the data cache when the `FirstUserTask` has just been created, and just after it has been activated for the first time.
- **(iii)** Repeat (ii) when the `FirstUserTask` creates the first task. Assume as many arguments and variables local to the kernel as you think is reasonable.
- **(iv)** Assuming tasks have no arguments and no local variables how many task descriptors can be squeezed into the data cache under best conditions.

1.b. On the stack of the kernel. One method of storing task state is to push it onto the kernel's stack. In this part of the question we suppose a kernel that does so.

- **(i)** Draw the kernel stack immediately after `FirstUserTask` is created during kernel initialization and just after it has been activated for the first time.
- **(ii)** Draw the stack of the kernel just after `FirstUserTask` re-enters the kernel to Create the first task, `First`, in the program to be run, and again immediately after the task has been created.
- **(iii)** Assuming that `First` is the same priority as `FirstUserTask`, show the stack immediately after the next task activation.
- **(iv)** Suppose that whatever task was activated creates Second, equal in priority to `First`. Draw the kernel stack, before and after creation.
- **(v)** What happens in the long run if no more task creation occurs? If much task creation occurs? If much task creation and destruction occurs?
- **(vi)** Can these problems be overcome?

1.c. What you did in your kernel. If you followed the instructor and TAs you chose to put the task state mostly on the user stack.

- **(i)** Some task state must be kept in the kernel. How much was kept in your task descriptor?
- **(ii)** For each item of state in your task descriptor, why did you include it?
- **(iii)** In retrospect would you change what's in your task descriptor? If so, how? If not, why not?

---

* When the complete state of a task is stored in a kernel data structure, the data structure is often called the 'task block'.

Do Two of Questions 2–5.

Question 2. Debugging

Some bugs are quick and easy to remove; others are slow and hard. There are many ways in which the two types of bugs differ, of which the following are a small sample.

1. Easy bugs occur frequently; hard bugs occur infrequently.
2. Easy bugs occur predictably; hard bugs occur irregularly.
3. Easy bugs cause an immediate crash; hard bugs cause a crash only after much subsequent execution.
4. Easy bugs display the same symptoms every time; hard bugs always have different symptoms.

2.a. Symptoms of bugs. Above, four qualities that make bugs easy are listed, and four that make bugs hard. Explain, with examples, why these characteristics are associated with either ease or difficulty.

2.b. IRQ bugs. Hardware interrupts can happen between any two instructions when interrupts are enabled. In your OS interrupts are *never* enabled inside the kernel, and *always* enabled outside it. Thus, a fault-free kernel depends on having a fault-free context switch for every pair of adjacent instructions in user code.

A bug is hard if it occurs between a pair of instructions that is not very common in the code, and if the kernel bug only occurs when a hardware interrupt occurs between that particular pair. Then the bug occurs only after significant running time (several minutes, for example), and produces non-deterministic symptoms. Two such possible bugs were described in class. Describe the circumstances (instruction pair and subtle kernel bug) in which they occur and typical behaviour between their occurrence and a subsequent crash. Think up a third example and describe it.

2.c. Link register bugs. In describing context switches I pointed out that there are three separate link registers, and that it's essential to put them back into the right places. For each of the three, describe its function, where and when you get it, and where and when you put it back. There are three ways to pair up a set of three things. For each possible pair explain the future execution of the program when that pair of registers is interchanged.

Question 3. Auto-calibration of Deceleration and Acceleration.

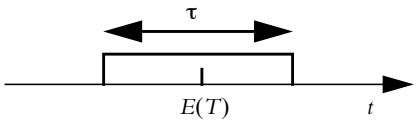
In class, I recommended so-called dynamic calibration for velocity, in which every sensor report is used to update the velocity estimate for the current speed. There are many other aspects of calibration that can be improved in the same way. In this question such techniques are gathered together as auto-calibration. The essence of the term auto-calibration is that it uses data collected for other reasons to improve your calibration.

3.a. Deceleration. There are many times during the demo when you stop a train. Most of these times the train passes over a sensor while stopping, giving you information about where the train is at a particular time when it is in the process of stopping. (Your project should be collecting this information automatically because you can use it to refine your estimate of the train's location when it stops completely.)

- **(i)** Describe in words how you would use this information to improve your velocity/time model of the train stopping.
- **(ii)** The velocity/time model requires a data structure. Give C pseudo-code for one that is suitable for ongoing update.
- **(iii)** The model also requires an algorithm that updates the model using the new data point. Give C pseudo-code for a suitable algorithm using the above data structure.

3.b. Acceleration. At a more advanced level you might want an auto-calibration model for trains accelerating from one non-zero speed to another. Repeat part **2.a.** to accomplish this task.

3.c. Probability distributions. In class we thought about the probability distribution, shown in the figure to the right, for the time at which a sensor report is registered in your program, its time-stamp. The probability distribution is centred on the expected value, $E(T)$, and (almost) rectangular, with width equal to the period of sensor polling, τ. You can use this property to improve the precision of a velocity/time model of acceleration/deceleration. Describe how you would do so. (Equations are acceptable in your answer.)

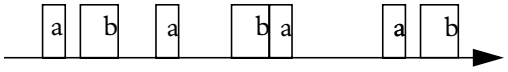


Question 4. CPU Congestion.

Although this question refers to part **3.c.** of Question 3, it is not necessary to do Question 3 in order to do this question, only to read it. One rule of thumb in the trains project is that the idle task should be running more than 90% of the time. This question explores the reasons behind that recommendation.

In general, we like the CPU to be inactive when an interrupt occurs so that the action needed in response to the interrupt occurs as quickly as possible. It is, of course, impossible to ensure this condition because the sources of interrupts are not synchronized. Therefore, the CPU is occasionally busy processing a previous interrupt when a new interrupt occurs.

Thus, the execution profile of your project resembles something like the time-line to the right, with the CPU 100% busy for a short time after each interrupt occurs. (The figure shows two activities running in response to periodic interrupts. Assume that the periods are incommensurate and that the computation will conflict from time to time.) In each burst of activity the first part is the running of high priority tasks associated with low-level input processing, followed by lower priority tasks that integrate the input into the state of the program and calculate the response to the interrupt. Transmitting the response through output devices then occurs sporadically, handled by high priority tasks. In this question we try to give this vague description a quantitative foundation.



As mentioned at the beginning of the exam, your answer should be based on a reasonably complete version of your project, such as a complete train control 2 result.

4.a. Enumerating interrupts. Most of the interrupt processing in your project occurs in response to one of six interrupts:

1. input ready from the train UART,
2. transmit buffer empty from the train UART,
3. modem flags changed from the train UART,
4. input ready from the terminal UART,
5. transmit buffer empty from the terminal UART, and
6. counted through zero from the timer.

For each of these interrupts, estimate

- **(i)** the maximum frequency at which it can occur (in occurrences per second),
- **(ii)** the average frequency at which it occurs (in occurrences per second),
- **(iii)** an estimate of the amount of processing it requires (in microseconds), and
- **(iv)** the deadline (in milliseconds) required to keep the activity running at 90% of its possible bandwidth.

1. For each answer give reasoning and/or calculations to support your answer.
2. You may want to apportion your answer among several different computations called for by the interrupt. For example, when transmitting train commands, there may be a significant difference between the first byte of a command and the second, or between a sensor poll command and a train speed command. If you do so give the answer as "x% this, y% that".
3. The numbers you give should be consistent with the idle time measurement for your project.

4.b. Coinciding interrupts.

- **(i)** Using the train transmit UART interrupt and the terminal input UART interrupt estimate the increase in response time given the priorities described above. Explain your answer as quantitatively as possible.
- **(ii)** The train transmitter UART might be transmitting a switch command to a turnout. How much competing CPU usage would be necessary to delay switching by 50 milliseconds?

Question 5. Destroy.

If you were building a kernel to support an application living as long as the applications in the Voyager spacecraft (37 years and counting) you would have to face problems that are not important in CS452: memory corruption and failure, device failure, and so on.* Using a kernel like yours, which is very different from the real-time executive on the Voyager spacecraft, a `Destroy()` primitive is most likely the way to recover from such errors. One might imagine implementing the Idle task to search the kernel data structures, looking for inconsistencies. When a pathological condition is discovered, one option for correcting would have the Idle task using the `Destroy()` primitive to remove one or more tasks, followed by recreation of the task(s) using the `Create()` primitive. This question asks you to consider several decisions you would have to make when designing and implementing `Destroy()`.

In all the parts that follow give reasons along with descriptions. Examples almost never decrease your mark.

5.a. Destroy(). `Destroy()` should leave the kernel in a consistent state with the destroyed task never running again. Below we call the inactivating a task.

- **(i)** List all the internal adjustments that the kernel must do to ensure consistency with the task never running again. Recovering resources is covered below.
- **(ii)** You implemented `Exit()`, a destroy-like primitive in the first kernel assignment. Describe how `Exit()` is related to `Destroy()` and `Suicide()` (part **5.b.**).

5.b. Suicide(). One might the Idle task a special kind of assassin. It would `Send()` a message to the task to be destroyed asking it to call the `Suicide()` kernel primitive. This is possible in some system states and not others.

- **(i)** Describe when `Suicide()` could work and when it can't.
- **(ii)** Assuming `Destroy()` to be available how would `Suicide()` be implemented?

5.c. Children. It is common to destroy the children of a task as well as the task itself.

- **(i)** Describe when destroying the children is a good idea.
- **(ii)** Give an example of a task configuration when destroying the children is not desirable.
- **(iii)** Because your system has an upper limit on the number of live tasks, you can bound above the amount of time required to permanently inactivate a task and its children. Calculate the worst case running time for inactivating a task and its children.

5.d. Resource reclamation. If there is to be much destroying of tasks, then resources must be reclaimed for future use.

- **(i)** Some resource reclamation can be done by the kernel. What are the resources and how are they made available for future use?
- **(ii)** If memory is to be zeroed, you probably don't want the kernel doing that. How might it be done?
- **(iii)** There are resources about which the kernel knows nothing, track reservations for example. How might these resources be freed?

---

* Providing a new program when bugs are discovered, which was done for the Mariner spacecraft, doesn't work very well: signals beyond Saturn are too weak.