University of Waterloo

cs452 – Final Examination

Winter 2015

Student Name:	____________________________________
Student ID Number:	____________________________________
Unix Userid:	____________________________________

Course Abbreviation:	cs452
Course Title:	Real-time Programming

Time and Date of Examination:	16.00, 14 April, 2015 to
	18.30, 15 April, 2015.
Duration of Examination:	26.5 hours.
Number of Pages:	7.

Rules of the Examination.

1. Do one of questions 1 & 2 plus two of the remaining 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). If PDF is inconvenient then I accept plain text (whatever.txt) but you will have to stretch a little to make diagrams. 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 **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 must make 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 **7.**,above, once more.

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

Do one of Questions 1 & 2 and

two of Questions 3, 4, & 5.

Question 1. 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.

1.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 **1.b.**).

1.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?

1.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.

1.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.

Question 2. Performance Analysis of your Kernel.

This question concerns two aspects of your kernel, Send/Receive/Reply and AwaitEvent. (Feel free to run your kernel to get estimates for the values required in this question.)

2.a. Send/Receive/Reply (SRR). In the second part of the kernel you measured the time taken by SRR for small and large messages with optimization turned off.

- **(i)** What were your times?
- **(ii)** During a system call three things occur, context switch(es), copying memory, and manipulating kernel data structures. Estimate the parameters A and B in the equation

$$SRRTime = Am + B,$$

  where m is the amount of memory copied in bytes. Answers should have units. Give your reasoning.

2.b. AwaitEvent. Assume that an interrupt occurs while execution is in the kernel processing an AwaitEvent system call on the interrupt's event. Between a call to AwaitEvent and its return the three things mentioned above, (ii), may occur, context switch(es), copying memory, and manipulating kernel data structures. If you measured the minimum call/return time for AwaitEvent, give the result, otherwise estimate it. Estimate the amount of time spent in switching context, copying memory and manipulating kernel data structures.

2.c. Context Switches. Counting the number of context switches in **2.a.** & **2.b.**, you can estimate the time for a context switch.

- **(i)** What is the context switch time?
- **(ii)** Look at your kernel code and consider whether the time required to run its instructions is commensurate with above time. Give numbers and explain your result.
- **(iii)** Not counting turning on the caches and compiler optimization, what did you do after the third kernel assignment to speed up your context switch? Explain your answer. Estimate the speed up.

2.d. Caches. Assume that both instruction and data caches of the ARM are lockable. Explain how you could use cache locking to improve the time taken by your context switch. Estimate the execution speed-ups when you turn on the caches, both with and without locking. Give as many concrete details as you can, such as how many cache lines to lock, what to put in the locked cache lines, the effect of a smaller cache on performance of user code, and so on.

Question 3. Short Moves.

In class I proposed the following method of making short moves starting and ending with the train stationary.

1. Give the train a `speed n` command.
2. Wait t seconds.
3. Give the train a `speed 0` command.
4. When the train stops measure how far it went.
5. Repeat the four steps above until you have values that span the distances you cannot handle by stopping from constant velocity.
6. Invert the table to get an interpolated function you can use to map the distance you want to go into the time you should wait.

3.a. Sizing the short move table. In step 5 above you must estimate the smallest distance at which you can stop using the constant velocity stopping distance. For your project what was that distance. (Estimate conservatively; give in full the reasoning behind your estimate; if you did not implement short moves in your project do this part using whatever calibration measurements you did. If you did not make any calibration measurements make up *plausible* values.)

3.b. Filling in the short move table. Once you know the range of distances you must decide the granularity of your table, the times you will use to sample.

3.c. Measuring acceleration using short moves. In this part assume that when you give a `speed n` command the train's velocity follows the following piecewise cubic function:

$$v(t) = \begin{cases} at^3 & 0 < t < t_0/2 \\ a\left(\frac{1}{4}t_0^3 - (t - t_0)^3\right) & t_0/2 < t < t_0 \end{cases} .$$

The train reaches its constant velocity $v(t_0) = \frac{1}{4}at_0^3$ at $t = t_0$. Explain how to use the short move table to estimate a without additional measurement. (Be practical, which means do no more work than you need to get an answer that is 'good enough'.)

3.d. Universal acceleration calibration. Suppose the cubic function is universal, in the sense that by rescaling a with t_0 constant you can get the profile of acceleration and deceleration from any velocity v_0 to any other achievable velocity v_1. Describe how you would test this supposition. Describe how universal acceleration would allow you to run the train at an arbitrary average velocity V. Describe how you could run your demo calibrated for any train, even a brand new one.

Question 4. Deadlock Avoidance.

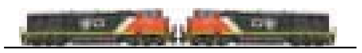
This question is about what you implemented, or would have implemented, in your project to avoid deadlocks. The first three parts present you with scenarios where deadlock is possible, and will actually occur if your software neither avoids or remedies deadlock. In each case you should describe

- how your software detects a possible or existing deadlock,
- what the software does to solve the deadlock, including other tasks that are involved in solving the deadlock, and
- what will happen over the next thirty seconds or so.

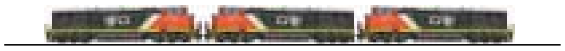
During the next thirty seconds there is usually more than one scenario.

Hints.

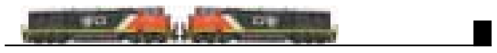
1. The content of messages sent is relevant.
2. All trains are trying to proceed with the orange end in front.
3. The scenario of trains travelling when they start moving is usually not unique. Indicate the variations that are possible.



4.a.. In the picture to the left two trains have stopped nose-to-nose, one centimetre apart on a track that forms a loop. As drawn, neither train can proceed to its destination because the other train is in its way.



4.b. In the picture to the left two trains have a third trapped between them on a track that forms a loop. As above, none of the trains can proceed in the absence of software that detects and remedies the deadlock. (Assume that the two trains on the right are travelling independently of one another.



4.c. The picture to the left has the trains stopped in the same positions as in **4.a.**but the track is not a loop. It ends at a bumper, indicated by the black rectangle. The train on the left is trying to travel in to the bumper; the train on the right is trying to leave.

4.d. Generalizing. Deadlocks similar in structure to the ones above can occur within an application program in the absence of an external world interacting with the program. When we add software to detect and resolve the deadlock, we are usually turning deadlock into livelock. Your kernel, at least in theory, can detect deadlocks and do something to resolve them. (In practice, detection is usually carried out by the idle task, which is given read permission on the kernel data structures.)

There are many deadlock conditions: deadlock detection tends to add code each time a new deadlock condition appears. Ideally we would have a general description of deadlock that catches all conditions; code would shrink along with debugging time.

(i) Describe the best general deadlock condition you could program into the Idle task. Explain what makes it good.

(ii) Do the same for deadlocks that arise among trains.

Question 5. Sensor Attribution.

Many students find that the 'sensor attribution' code they created for the first train control milestone (FTCM) does not generalize easily to what they need for the second milestone (STCM). As an instructor this frustrates me because it wastes student time, which is in short supply. This question asks, in essence, 'How could one teach what to do for the FTCM so that the code created would generalize easily to the SCTM?'

I see the big picture as follows. For the FTCM it is necessary to classify sensor reports into two classes: ones produced by a train, which I call genuine, and others, which I call spurious*, that are not produced by a train. Classification requires a well-defined criterion separating genuine from spurious, which is implemented in the code responding to sensor reports. For the STCM, the criterion is generalized to sort sensor reports into three categories. This question focusses on the criterion, assuming that when it's well enough defined implementation follows straightforwardly.

In your answers to the questions that follow, I expect to see examples of success and failure, reasons why your criterion succeeded or failed, ideas about how to do better and generalizations to other domains.†

5.a. FTCM.

- **(i)** What the criterion did you use for the FTCM? If you didn't create the criterion before implementing it, look back at your code and work out what you did.
- **(ii)** How well did your criterion work?
- **(iii)** Describe the reasons why you chose it.

5.b. STCM.

- **(i)** What the criterion did you use for the STCM? If you didn't create the criterion before implementing it, look back at your code and work out what you did.
- **(ii)** How well did your criterion work?
- **(iii)** How much of your FTCM solution remained in the STCM one? What does that tell you about the similarity of the STCM and FTCM problems?
- **(iv)**

5.c. Teaching. You probably noticed during the course that the lectures, which are very directive at the beginning, gradually become less directive. By the end of the course they mainly describe problems, expecting you to solve them on your own. When I notice an ongoing problem late in the course, like the one discussed in this question, I should not be thinking what solution I should teach, but what problems I should teach when. For example, I have noticed while writing this question that the phrase 'sensor attribution' should occur fairly prominently in my FTCM teaching.

- **(i)** If you were teaching the project part of the course how would you describe the sensor reading problem for FTCM so that students would get a solution that is generalizable to STCM?
- **(ii)** Another problem I notice some terms more than others is groups that put too much effort into route finding and not enough into calibration when working towards FTCM. What is the right balance? If you were teaching how would you teach so that students would consistently get the right balance? (As most teachers know simply telling students, 'Calibration is more important; you should be spending most of your time on calibration,' doesn't actually work very well.)

* **Spurious.** not genuine, not being what it pretends to be, not proceeding from the pretended source (Concise Oxford Dictionary).

† Computer-mediated sensor attribution exists in many domains: the timing of green arrows for turns at traffic intersections, moving objects in computer vision and radar for air traffic control, among many others.