RULES OF THE EXAMINATION.

1. Do questions 1 & 2 plus one 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.)
5. Put your name, student number and userid on every page.
6. Your answers should be submitted by e-mailing them to me at wmcowan@cgl.uwaterloo.ca.
7. Read 7., above, once more.
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 standard units. If your unit is ticks, for example, translate it into milliseconds using the size of tick in your kernel.
10. There may be places in the examination where you will want to make assumptions. Do so, being certain that you explain your assumptions and how they are related to the question you are answering.
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 Questions 1 & 2 and one of Questions 3, 4 & 5.


In the first part of the kernel you had to design the memory layout of your kernel, which probably stayed pretty much the same during kernel and program development. Many students ask how much memory a task needs, and I give them the standard answer, ‘It depends.’ You have had a lot more experience with programming tasks by now, and maybe it’s time to think about how one might estimate the memory needed for a task.

This question asks about bounds on the size of the stack. (When the question says ‘bounded’ it means bounded by a finite number. You may assume that the number of tasks in the system is bounded by a finite number, $N$, which is the size of the array of task descriptors.)

1.a. A Courier. A courier has a very simple skeleton.

```plaintext
init();
for( ; true; ) {
    Send(srcTid, REQ, message);
    Send(dstTid, message, ACK);
}
```

(i) Draw a sketch of the data elements on this courier’s stack when it is as big as it ever can be.

(ii) Give the size of the stack, making realistic assumptions about the size of the data elements it uses.

1.b. The Time Functionality of a Clock Server. The skeleton of the time functionality of a clock server looks like this.

```plaintext
init();
for( ; true; ) {
    Receive(reqTid, req);
    switch(req.type){
    case CLIENT:
        Reply(reqTid, time);
        break;
    case NOTIFIER:
        time++;
        Reply(reqTid, ACK);
        break;
    }
}
```

(i) List the elements that are on this server’s stack when the stack is at its biggest, giving the maximum size of each and the total size of the stack.

* ‘Skeleton’ means that I have omitted error-detection and other stuff leaving only the overall structure of the task.
1.c. The Delay Functionality of a Clock Server. The delay functionality of a clock server is a little more complex. Its skeleton looks like the following.

```c
init();
for( ; true; ) {
    Receive(reqTid, req);
    switch(req.type){
    case CLIENT:
        insert(ordered-list, {reqTid, time + req.delay});
        break;
    case NOTIFIER:
        time++;
        for(h=extract-head(ordered-list);
            h.time < time;
            h=extract-head(ordered-list) ) Reply(h.tid, ACK);
        insert(ordered-list, h);
    }
}
```

(i) List the elements that are on this server's stack when the stack is at its biggest, giving the maximum size of each and the total size of the stack.

(ii) Give a general rule for bounding the stack.

(iii) This bound implies a bound on the run-time of computation required to handle any request. Explain.

1.d. The Big Question. The first two parts of this question shows that it is easy to bound the size of the stack when there is no looping or recursion. The last part shows that it is possible to bound the size of the stack in the presence of some types of looping, which could take the form of recursion.

(i) There are, of course, examples of looping and recursion in the presence of which the size of the stack cannot be bounded. Give and explain an example. (Hint. Think about calling functions.)

(ii) Give a general rule of thumb that specifies when a task has a bounded stack.
Question 2. Simulating the Train Set.

Every few terms one or another group builds a simulation of the train set. Doing so requires a model of the actual train set similar to the one that you build when you measure properties of trains responding to commands. This question explores types of simulation that you could build. When doing this question think about which aspects of debugging a train project would be usefully supported by the simulator. Assume that the final demo must be done using physical trains with realistic unpredictabilities. Make any assumptions you need to fill out each of the simulations, being sure to state your assumptions explicitly. The simulation types described below are ordered by increasing amount of implementation effort.

Please notice that the simulations described in this question are not the model your program builds when it runs the trains on the track. They are autonomous programs connected to your train program through the UART that is normally connected to the train set. Your program builds it internal model of the train by creating a calibration based on input received from the simulation.

2.a. Naive Simulations. A naive simulation (NS) makes naive assumptions about trains like the assumptions you made for the first milestone: trains have zero length; trains get the steady state velocity instantly when given a speed command; trains’ stopping distances are always the same; the stopping time is always the same; the lag between a command being given and received by the train is zero, and so on.

(i) Give two bugs that would be easy to detect and remove using an NS connected to a program running on the ARM CPU. Give specific details of the bugs, how they occur in the NS and how you would use the NS to identify and remove them.

(ii) Explain problems you might have when implementing bug fixes found in the simulator to programs controlling actual trains.

2.b. Exact Simulations. An exact simulation (ES) has train set properties – acceleration, velocity, lag times, length of the train – that are fully deterministic, so that the part of the track occupied by a train in the simulation is known at all times. When, for example, a stationary train receives a speed command, it accelerates according to a deterministic acceleration function, then travels at a deterministic velocity. And as it travels, sensor reports are provided at the exact time that the pick-up of the train reaches a sensor. These properties, while exact, may not be the same as the properties of actual trains.

(i) Give two bugs that are easy to detect and remove using an ES connected to a train program running on the ARM CPU. Describe the symptoms provided by the ES and how you would use them to remove the bugs from your program.

(ii) How would the symptoms of the bugs in the simulation differ from the bugs as they occur with the actual train set.

2.c. Stochastic Simulations. A stochastic simulation (SS) is like an exact simulation but takes into account variability in the properties of the train set. For example, lags between when a command is given and when the train set carries it out, such as the lag between the request to switch a switch and the actual switching of the switch, may vary from one request to the next. The most important stochastic effects in the simulation are probably the uncertainties in the measured timings of sensor reports. An SS, like an ES has idealized train behaviour, such as acceleration and velocity functions.

(i) An SS, unlike an ES, varies from run to run of the simulation, even though its input is the same, which gives it new properties that are similar to those of the train set and that reveal further weaknesses of a train control program. Describe two in detail, giving their observable symptoms, and describing how the behaviour of an SS differs from the actual train set.

(ii) In class I talked at some length about the causes and properties of time variation in sensor reports. Discuss how you would use that material in building an SS.

(iii) How would the behaviour of dynamic calibration differ between an SS and the actual train set?
2.d. **Measured Simulations.** The simulations described so far use idealized properties for trains, such as the functions used to model accelerations in Question 5. A measured simulation (MS) uses calibration measurements of actual trains, switches and sensors, possibly described as parametrized functions.

(i) An MS might not save any effort at all. The time spent preparing data for the simulation might exceed the time spent working directly with the actual train set. For example, the measured properties of a given train vary over time. Evaluate the costs and benefits of a MS under these circumstances. What value has the simulation under such circumstances?

(ii) If an MS has a cost that exceeds the benefit, there is probably a sweet spot among the types of simulation where the benefit of simulating minus its cost is maximized. Where do you think it is? Give your reasoning.
Question 3. Saving Context.

When a task is not active its registers must be saved somewhere. This question is about the various possible locations for storing the saved registers. In this question, and the remainder of the exam, you are often asked for reasons

3.a. The Stack of the Task. In most cs452 kernels, context switching code places the register values of tasks entering the kernel on the stack of the task to which they belonged. In class I gave you two reasons why this is a good idea, and one reason why it isn’t a bad idea. Call this three reasons why it’s a good idea. There are other reasons why it’s a good idea; feel free to use them in your answer if you prefer them to the ones I mentioned. (Hint. Treat stacks as stacks throughout Question 3, which means that you can push and pop, but not reach into the stack at random places.)

(i) Give three reasons why you think putting the registers on the stack of the user task is a good idea, and explain why. You can explain why one of my reasons is not actually a good reason if you wish.

(ii) Quite likely you pushed other parts of user task state onto the user stack. What things did you store there? Why do they belong on the user stack?

(iii) There may be state you stored elsewhere. Tell me what it was, where you stored it, and explain why you stored it there.

3.b. In the Task Descriptor. Real-time executives commonly store user state in the task descriptor of the task, which is found in the kernel data space. In this course you were strongly discouraged from doing so. (Hint. Your answers should estimate the size of data you are storing.)

(i) Explain why doing so was discouraged in cs452.

(ii) Why do you think that storing the state of user programs is so common in embedded systems? (You might want to look at the ARM processor architectures of small scale CPUs.)

3.c. On the Kernel’s Stack. Another option for saving user state is on the kernel’s stack, which might seem attractive because kernel instructions store the values in registers, then load them back into the registers when the task is next activated. (Remember that a stack should never do anything but push and pop.)

(i) Is there any time during execution when there would be two instances of a task’s registers on the kernel stack? Why?

(ii) Saving on the kernel stack is a bad idea. Explain why, giving diagrams showing the state of the kernel at different times during the execution of several tasks.

3.d. The Instructions that do the Saving. This question has implicitly assumed that kernel instructions save the registers, because that’s the way it is in cs452 kernels. In contrast, most compilers share register saving between the code of the function caller and the code of the function itself.

(i) Explain why this is possible for functions and not possible for system calls. In your answer take into account that it is possible for some system calls and not for others.

(ii) One option would be subdividing system calls into two different types and, different instructions for each. Give the pros and cons of such a design.
Question 4. Hardware Protection of Tasks.

In a shared computer system user programs must be prevented from damaging the operating system. Equally, user programs should be prevented from damaging one another. Commonly, this is accomplished using hardware protection of memory. You certainly saw the hardware component that implements protection, the memory management unit (MMU), in your second year hardware course, and a variety of ways of using the MMU in your third year operating systems course.

Your kernel probably did not program the MMU; the bootloader set the MMU to give you a convenient view of memory, which you used in your kernel. You may want to refer to the MMU section of the ARM processor manual when answering this question.

4.a Handling an Illegal Access. When the MMU in our ARM CPU detects an attempt to access protected memory it causes a data abort exception, which should trigger code that cleans up the problem. For example, most versions of Unix kill a program that tries to access protected memory, providing only a cryptic message like ‘Segmentation violation’.

(i) Suppose you wanted to do what Unix does when a data abort exception occurs. You would need to enter the kernel, print ‘Segmentation violation’ on the terminal and exit. (I am assuming that you would immediately debug the problem and start a new bug-free execution.) List the sequence of steps that occur as this happens, including special values that are needed to support the steps.

(ii) To help the person debugging the program the kernel might do a little more than just printing a generic error message. List three pieces of useful information that the kernel might provide, explaining how you obtain them and why they would be useful for debugging.

4.b Protecting the Kernel. A simple static protection mechanism might protect all kernel memory from access by user tasks.

(i) List three distinct blocks of memory that should be protected from access by user tasks, explaining what is special about each one.

(ii) Describe how you would set up the MMU to provide this protection.

(iii) This protection scheme catches many stray pointer errors. What is the most common value of an uninitialized pointer?

4.c Protecting Individual Tasks. Protecting the kernel helps, but it’s also common for tasks to interfere with one another. You may have experienced, for example, user task stacks running into one another. So here’s something you might want to do. When a user task is active arrange the MMU so that it can access only its stack. An attempt to access other memory would cause a data abort exception.

(i) Describe in words how you would set up the MMU to get this protection.

(ii) Describe how the MMU tables would change when a context switch occurred from a user task to the kernel, and from the kernel to a user task.

Sometimes a new table for the MMU is found in low memory, sometimes it’s found in the table lookaside buffer (TLB).

(iii) If you had fifty tasks, how often would the new table be found in the TLB?

(iv) Compare the TLB to the L1 cache. How much speed-up did you get in the message copying part of message passing when you turned on the L1 cache? What does this tell you about having to go to memory for a new MMU table?
Question 5. Short Moves.

In class I proposed the following method of making short moves starting and ending with the train stationary. Many of you adopted it in your projects.

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.

5.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.)

5.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.

5.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:

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

The train reaches its constant velocity $v(t_0) = \frac{1}{4}a t_0^3$ 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’.)

5.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.

$\left(9\text{ of }9\right)$