CS452 aka ‘The Trains Course’ – Final Examination

Spring, 2016

Instructor: Bill Cowan

Starting time: 19.00, 1 August, 2016.

Please e-mail your answers to the instructor at wmcowan@cgl.uwaterloo.ca before the ending time.

Things to remember when reading and writing this examination.

Make sure that you understand what is written below. Term after term more marks are lost to not following instructions than to any other cause.

1. Do Question 1 and any two (2) of questions 2–4. Doing the bonus questions is almost never a good use of your time: marks are added more quickly as you respond to other questions. It does, however, give you an opportunity to put the marker in a good mood, by your interest in something that obviously interests him, and by the scope for making him laugh.

2. You must work independently.

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. There may be places in the examination where you can say something clever by making assumptions. Do so, but explain your assumptions and how they are related to what you are saying.

5. I take it for granted that you know CS452 to be the REAL-TIME course. Pretty well every question requires an answer that takes time into account even though time is rarely mentioned explicitly in question descriptions.

6. Remember that in real-time programming we care most about deadlines, the time between an event occurring in the outside world and the time when the correct response must be completed. In real-time systems a too soon response is equally bad as a too late one.

7. Any time you can provide a quantitative estimate of performance you are gaining more marks! All numbers should have units.

8. Have a dictionary at hand when reading and look up any unfamiliar words.

9. Submit your answers by e-mailing them to wmcowan@cgl.uwaterloo.ca.

10. All questions are open-ended: most marks come from going beyond simple answers; explicit instructions are intended as prompts to get you started in the right direction. 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.

11. In the examination ‘your kernel’ means the kernel you actually created, not the kernel you wish you had created. Similarly, ‘your OS’ it means your kernel plus the other tasks (couriers, notifiers, servers) that provide the kernel API. On the other hand, ‘your train application’ means the application you planned to create in its final form.
Question 1. The GPIO Interface

Note. You must do this question. In addition, do two (2) of questions 2–4. Do bonus questions only after doing the best you can on the other questions.

Note. As you read through the exam questions you will, no doubt, find spelling and grammar mistakes. They are inevitable when typing the order of four thousand words. If you find them early in the exam e-mail them to me: I will correct the on-line copy and send out errata.

Now the question itself starts. When it asks you to write (pseudo-)code, please include on each line a short phrase saying what the code does.

Once upon a time a Fine Arts student asked me to help her build hardware and software for a position sensor that consisted of 16 pressure sensitive tiles, each 100 cm by 50 cm. The tiles were to be placed on the floor in front of an interactive painting. The device sensed the position of a viewer in front of the painting, and subtly changed the painting’s appearance as they moved around.

Each tile contained a micro-switch that shorted its outputs when there was weight on the tiles. As is usually the case, the switches were open-circuit when the tile carried no weight, and short-circuit otherwise. Each output was connected to the input pins of one bit of a sixteen-bit digital I/O adapter. The digital I/O adapter we used pulled up its inputs: when there was no weight on the a tile the corresponding input was asserted (1); when there was weight it was negated (0).

The digital I/O adapter had a data register like the data register of a UART – but sixteen bits wide, which could be read on the system bus, and asserted an interrupt if any of the bits changed. (The switches were debounced within the tiles.)

If you were to receive a similar request you could easily do so using Ports A and B of the GPIO Interface on the Cirrus EP9302 (Chapter 28). In reading the documentation there are a few things you might bear in mind.

1. The GPIO interrupt available in the ICU (Chapter 6) is GPIOINTR, interrupt 59. This interrupt is the OR of interrupts of each individual interrupt-enabled bit in the GPIO.

2. In the GPIO concern yourself with Data, Direction, Interrupt Enable, End-Of-Interrupt, Interrupt Status and Raw Interrupt Status registers; the other registers control aspects of the GPIO you may treat as irrelevant for responding to the tiles.

Part 1.1. Initialization

Before the Tile server can start providing reports to the application the GPIO and the ICU must be initialized. Give a sequence of ARM assembly language instructions that will initialize the devices.

Part 1.2. Implementation of GetTiles and GetTilesNow

The Tile server should support two ways for getting the state of the tiles: one non-blocking, GetTilesNow( ), and one blocking, GetTiles( ). The first is suitable for an application that wants periodically to poll the state of the tiles, calculating changes – if any – at regular intervals; the second is suitable for an application that responds one by one to a sequence of changes of tile state.
TileMap GetTilesNow(). GetTilesNow() responds immediately with the current state of the tiles as well as it can be known by the tile server. (TileMap is a bit-map with one bit per tile.)

TileMap GetTiles(). If there are changes that occur with no GetTiles() requests the server accumulates a buffer of changes, containing the state of the tiles at each change. Each time void GetTiles() is called it responds with the next value in the buffer. If the buffer is empty GetTiles() blocks until the next change occurs.

Implement in pseudo-code the FOREVER block of the Notifier and the Server that provide the two services described above to clients. (You need not give the code that wraps Sends to the Tile server.)

Part 1.3. Implementation of Shake

Now suppose that the artist locates a device that gives a tile as little shake whenever its input is asserted. She wishes to use it to give feedback to the viewer as he or she moves around the tiles. To save rewiring, the input to the shakers is multiplexed onto the wires that provide input from the tile sensors.

Activating shakers then requires changing the direction of the appropriate bits in the GPIO then asserting them. You will put this functionality into the Tile server, giving the application a function void Shake(TileMap tiles) that is a wrapper for a Send to the Tile server. Give pseudo-code that adds this functionality to the FOREVER block of the Tile server.
Question 2. User State in the Kernel

Do two (2) of questions 2–4.

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

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 they may cause.

Part 2.1. 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 practice is common in many embedded applications and recommended in many engineering OS classes.

In answering the following questions assume that the task descriptors are allocated as an array on the kernel stack, as they were in your kernel.

i. How big, in 32-bit machine words, would such a 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. Suppose you were to lock all the task descriptors into the data cache during kernel initialization. Using the number of task descriptors defined in your kernel how much of the data cache would you need? What impact would this have on the execution of your kernel and train application.

Part 2.2. On the stack of the kernel.

One method of storing task state is to push it onto a stack, the task state (TS) stack, in the kernel’s address space. To be specific, when the kernel starts executing after a software or hardware interrupt it pushes the state of the active task onto the TS stack. When the task is later re-activated the kernel pops its state off the TS stack and installs it in the processor. In this part of the question we suppose a kernel using a TS stack.

In this part we consider the TS stack to be a real stack: the only operations it supports are pushing and popping.

i. How task state is stored strongly affects task creation. How does this work? Consider the first user task (FUT). It must be created with its state on the top of the stack so that it can be popped off when FUT is first activated. Draw the TS stack and the ready queues (RQs) just before FUT is scheduled for the first time.

ii. Now suppose that the first thing FUT does is to create a task at a higher priority that immediately exits. Give a rule for where newly created task state should be placed on the TS stack and draw the RQs and the TS stack just before the second scheduling occurs. (Assume that exiting pops the stack.)

iii. Repeat the question above for a task created at a lower priority, generalizing the task creation rule used above. FUT exits after the creation allowing the lower
priority task to run.
iv. Repeat the question above for two tasks created at the same lower priority. The rule above should work without change.
v. Now change the created tasks from the previous part to that each executes pass before exiting. Show the RQs and the ST stack just before scheduling.
vi. Explain the basic incompatibility between a stack and the scheduling we do in our kernel.

Part 2.3. What you did in your kernel.
If you followed the advice of 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. Suppose you were to lock all the task descriptors into the data cache during kernel initialization. Using the number of task descriptors defined in your kernel how much of the data cache would you need? What impact would this have on the execution of your kernel and train application.
Question 3. Calibration

Do two (2) of questions 2–4.

Note. From your project you should be aware of typical values of quantities discussed below. The more you use these values in your answers, the higher your mark.

The train set has three problematic features that usually exist when programs control real-time real-world analog 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.

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 four aspects of train kinematics you may have measured:
1. the stopping distance for speed values you planned to use and the time interval required to stop,
2. the steady-state velocity of the train for each speed value you planned to use,
3. the distance a train moves during a short move and the time taken by the move, and
4. the deceleration and acceleration profiles from one speed value to another. (A profile tells you the velocity as a function of time after a speed change command is given.)

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

Part 3.1. The Calibrated System.

‘The details of calibration always depend on the controlled system, including the application task it performs.’ Explain this statement. Give concrete examples from your train application, and other applications from which it differs.

Part 3.2. The Three Problematic Features.

Give an example from your train application of each of the three problematic features listed above, and explain why it makes train control difficult.

Part 3.3. The Stopping Distance.

Suppose you know only the stopping distance of a train.

i. Give a precise description of what is included in the stopping distance/time as you defined it in your train project, explaining why each part is or isn’t important.
ii. Knowing only the stopping distance where on the track can you stop the train?
iii. Describe the sequence of events required to stop the train.

Part 3.4. Constant Velocity.

When you know the stopping distance and the steady state velocity of a speed you can stop the train at a pre-specified location most of the time.
cs452 : Final Examination

i. Describe in words the calculation required to stop the train \( y \) cm past the sensor at location \( S_x \) cm.

ii. What sequence of events occurs in the application program to make this stop occur?

iii. Under what circumstances is it not possible to stop at a predefined location knowing only the stopping distance and the steady state velocity for each speed?

Part 3.5. Acceleration.

Short moves and acceleration profiles have a great deal in common. In fact each can be determined from the other, at least in theory: for example, several groups calculated their short move times from acceleration profiles.

i. Describe in detail how to create an acceleration profile from a set of short move distances and times. Be explicit about any assumptions that you make.

ii. During the demos I saw many short move calibrations that were quite precise at short distance but seriously imprecise at longer distances. Many were the result of deriving short move parameters from an acceleration profile. Using the document I wrote about using the wrong model, explain why this might be so.
cs452 : Final Examination

**Question 4. Fluidity of a Railroad System**

Do two (2) of questions 2–4.

When railroaders talk about the operational health of a railroad, fluidity is usually a central concept. They see trains moving on the tracks as fluid moving in pipes or water moving in rivers. Trains flow into a classification yard, where they are broken down into parts by destination and reassembled into new trains with all cars having the same destination, and then flow out again. The yard is like a buffer; as long as there is enough space in the buffer trains move smoothly through it. But if the buffer fills, trains back up on the incoming track and everything stops because the cars you need to complete leaving trains can’t get into the yard: the system has frozen.

**Part 4.1. Theoretical limits**

The track configuration in the trains lab was deliberately created to challenge fluidity. Real railroads have kilometre after kilometre of track quite free of turn-outs, with occasional interlocks, where trains can move from one track to another, or vice versa, which is not much of a challenge for trains students.

Most projects reserve track breadth first, reserving a fan-shaped collection of track along all the routes that might be occupied in the future. We can make a little model of this system for you to explore. We assume the following.

- Reservable pieces of track (edges) are bounded by sensors, merges and branches (vertices). Merges and branches are turn-outs, the first seen as two tracks coming together into one, the second as one track splitting into two.
- Merges can be counted as sensors: that is, loops are removed from the tree by duplicating edges. They can also be ignored.
- Branches are a fraction $p$ of section divisions.

We now describe a process by which bigger reservations are constructed systematically from smaller reservations, creating the reservation fan.

1. A train sits on an edge, between two vertices. That edge is reserved, owned by the train sitting on it.
2. Forward, as seen by the train, points to one of the vertices, the **leading vertex**.
3. When the train wishes to move it asks for edges beyond the leading vertex. If the leading vertex is a branch it asks for two additional edges, otherwise one.
4. After the reservation has been granted either three edges have been reserved, with two leading vertices, or two edges have been reserved, with one leading vertex. The former happens with probability $p$, the latter with probability $1 - p$. Thus, the average number of reserved edges is $3p + 2(1 - p) = 2 + p$, and the average number of leading vertices is $2p + (1 - p) = 1 + p$.
5. As long as the train needs more free distance it continues to reserve track. Suppose that when the fan is $n$ deep the average number of leading vertices is $l(n)$ and the average number of reserved sections is $r(n)$. Then

\begin{align}
  l(n + 1) &= 2pl(n) + (1 - p)l(n) = (1 + p)l(n) \\
\end{align}

(4.1)

and
\[ r(n + 1) = r(n) + 2pl(n) + (1 - p)l(n) = r(n) + (1 + p)l(n). \] (4.2)

i. Check that (4.1) and (4.2) are correct for \( l(1) \) and \( r(1) \), noticing that \( l(0) = r(0) = 1 \).

ii. Solve the recursions, showing your work.

(I think that the answer is \( l(n) = (1 + p)^n \) and \( r(n) = (1 - (1 + p)^{n+1})/p \). At least in the limit as \( p \) goes to zero \( r(n) = n + 1 \) is correct.)

These numbers grow fast with \( n \), but not so fast as to be unmanageable on a track the size of ours.

Part 4.2. The quality of the model

We make theoretical models not because they are ‘correct’, but because they give an understanding of ideal cases. We extend this understanding to more complex cases by faith and analogy. When we make a model we must estimate how it falls short of the reality with which we must work in practice. The current model is both overly optimistic and overly pessimistic.

In considering the shortcomings of a model always recall what we are trying to understand. In this case, we are trying to understand why and when reservation start to interfere with one another in the specific track configurations in which we will demo our projects.

i. \textit{Overly optimistic.} The model is overly optimistic in neglecting the details of track geometry. Some sections of track are more likely than others to produce reservation conflicts. Explain. (Reservation conflicts are the primary reason why trains must wait, which limits fluidity.)

ii. \textit{Overly pessimistic.} The model is overly pessimistic in neglecting merges, which reduce the amount of track reserved. Explain.

Part 4.3. Using the model

The model abstracts away all details of how sections of track are attached to one another, the topology of the track. It does, however, retain some differences of how sections of track are joined together, \textit{viz} how branches create fans.

In addition, the model assumes that several sections of track are required for a reservation: that is, a train requires a distance to stop that is bigger than the average section length. This assumption is valid. However, it measures the size of a reservation by the number of sections it contains, abstracting away the large variability of section length.

i. How many section dividers of what kind are there on your favourite track? How many sections are there? What’s an appropriate value to use for \( p \)? (Please don’t send me e-mail asking me the right way to count: in this question I want to know if you can translate vague descriptions of counting into pointing at actual things, saying ‘One, two, three, . . . ’)

ii. What is your estimate for \( p \)? Explain.

iii. What is your estimate for the number of sections required to stop a train at the speeds and on the sections of track that you used in your project? Explain.

August 1, 2016
iv. What is a typical reservation size, in sections, required by a train in your project?

v. How many trains do you think you could keep moving on the track in the lab?

This number is not simply the result of algebraic manipulation of the numbers you put down above. First of all, it is likely to be a range, not a single number. Second, it will be the result of thinking and evaluating as well as of calculating. Tell me what you are thinking in your answer.

The purpose of making models like the one in this question is to limit ignorance, which is often worse than stupidity. (What’s worse than stupidity is not ‘limiting ignorance’ but ignorance itself.) Computer graphics students from time to time draw at Alpha Centauri, which is in the logical display space, but not the physical one, then wonder for days why they can’t see their drawing. The trains course equivalent is carefully designing a project that will be able to run twenty-five trains simultaneously. When models like this one prevent errors like those ones they are worth their weight in gold. (Which is?)
Bonus Question 1. Tile Hardware

In the introduction to Question 1, I introduced two hardware concepts that may be unfamiliar to most of you: ‘pulling up inputs’ and ‘debouncing switches’. You may also have noticed that switch debouncing and default level are options of GPIO on the Cirrus EP9302. Explain the meaning of each, and why it is important to the application described.