CS 452 P1

Names: Robert Elder, Christopher Foo
ID #: 20335246, 20309244
Userids: relder, chfoo

Date due: July 9, 2013

Running

The executable is located at `/u/cs452/tftp/ARM/relder-chfoo/p1-submit/kern.elf`.

The entry point is located at `0x00045000` or `%{FREEMEMLO}` It *must* be executed with caching enabled:

```
load -b %{FREEMEMLO} -h 10.15.167.4 ARM/relder-chfoo/p1-
submit/kern.elf
go -c
```

Commands

tr TRAIN SPEED Set the train speed.

rv TRAIN Slows, stops, and reverses train. The final speed is hard coded to 5.

sw SWITCH DIRECTION Changes the turnout direction. DIRECTION is either S or C.

q Quits the program.

map NAME Sets the current track. NAME should be A or B.

Figure 2 and Figure 3 show different map configurations.

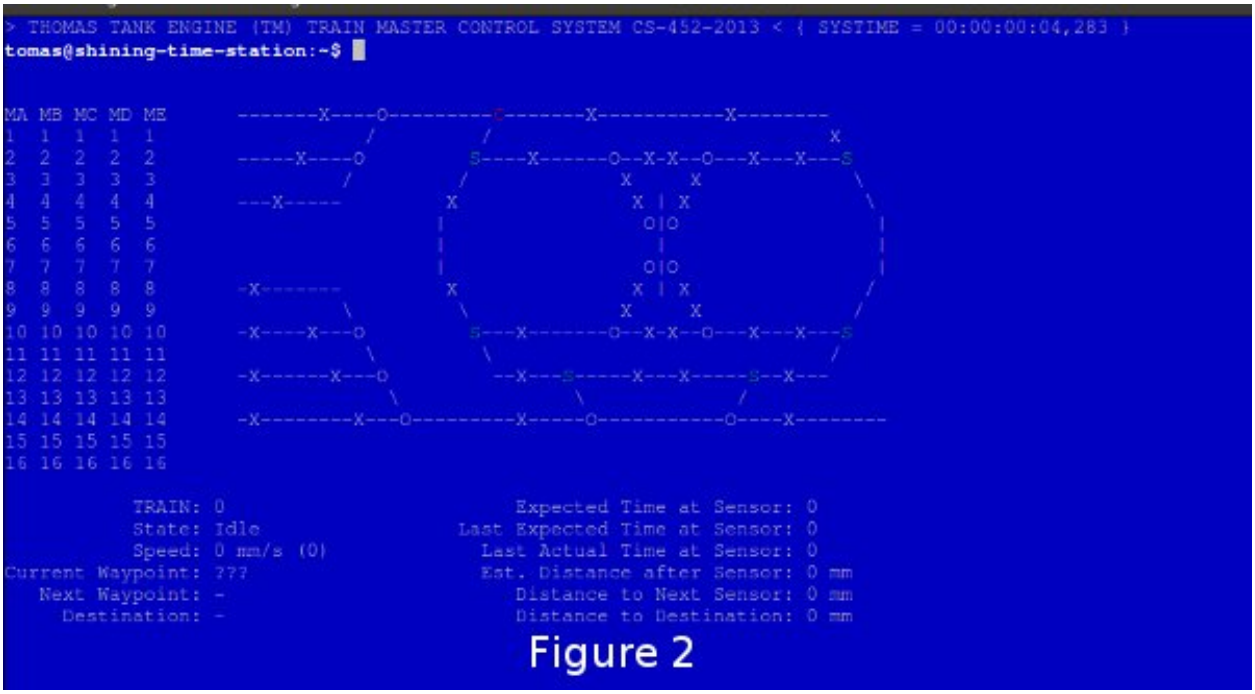

Figure 2

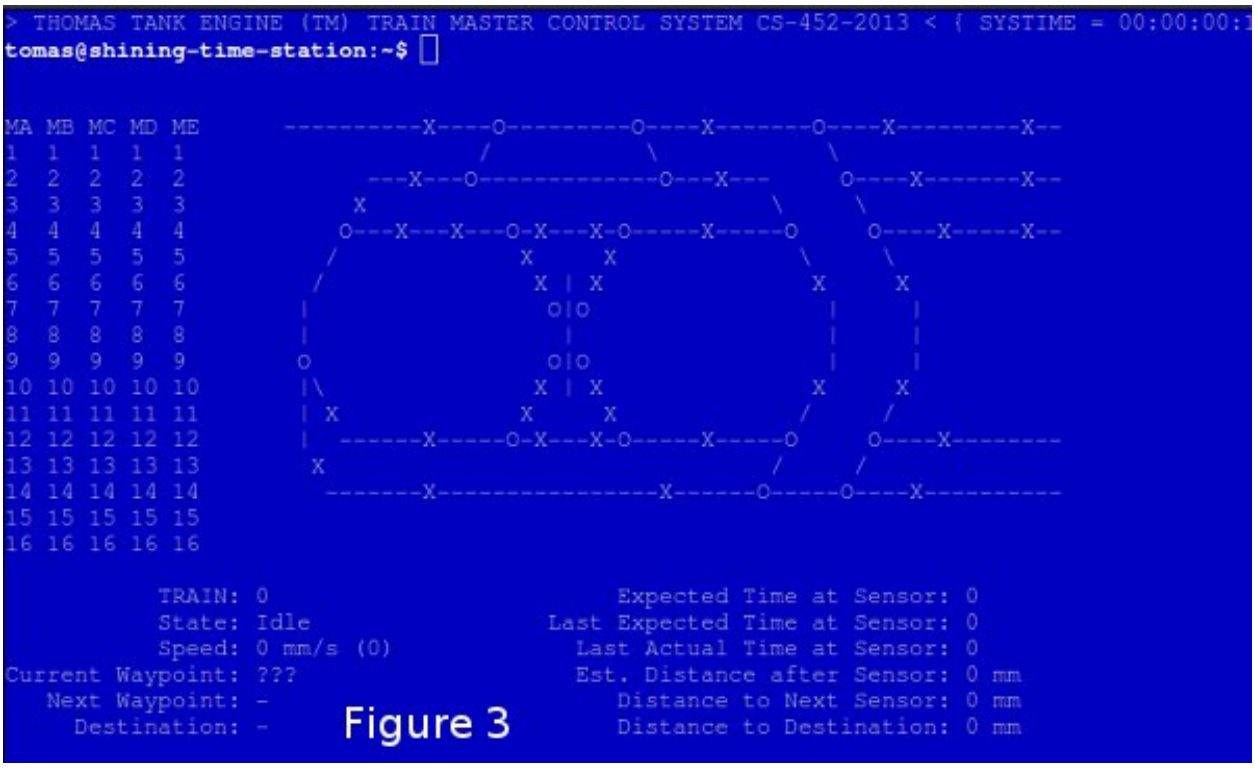

Figure 3

go TRAIN Begins the train route finding process. The train should start up, find position, and go to a random destination.

gf TRAIN Like `go`, however, this make the train go forever by running `go` in an continuous loop.

Pressing 'z' will cause the program to dump out a list of tasks information and statistics. This is considered a debug operation, and as such it can cause future instability in the program.

Description

Kernel

- Caching improves the performance of the program and will be mandatory for this deliverable. We have finely tuned the duration of a time slice to be no longer than 700 microseconds. Running the program without caches would require re-tuning of the time slice to prevent all CPU cycles being burnt up doing context switching. The extremely small time quantum of 700 microseconds ensures fast responsiveness to train input and with the user interface.
- Interrupts have been completely refactored to increase stability with train communication. This was done because we were previously attempting to send on the TXFE interrupt instead of waiting for CTS to be asserted. We now listen for the modem status interrupt and correctly attempt to send information to the train only when CTS has been asserted after a de-assertion.
- Much investigation and refactoring was done to be sure that we will not miss any data when communication with the train. A simple busy waiting test program was created to evaluate the timings and state transitions that happen when communicating with the train. A diagram is shown in Figure 1 that presents the empirical timings of io flag assertions when communicating with the train. One interesting finding is that the amount of time it takes for data to be returned from a specific sensor module can vary by about 5 milliseconds, however the time between bytes of data sent back from the train controller is very constant, at about 4.87ms.

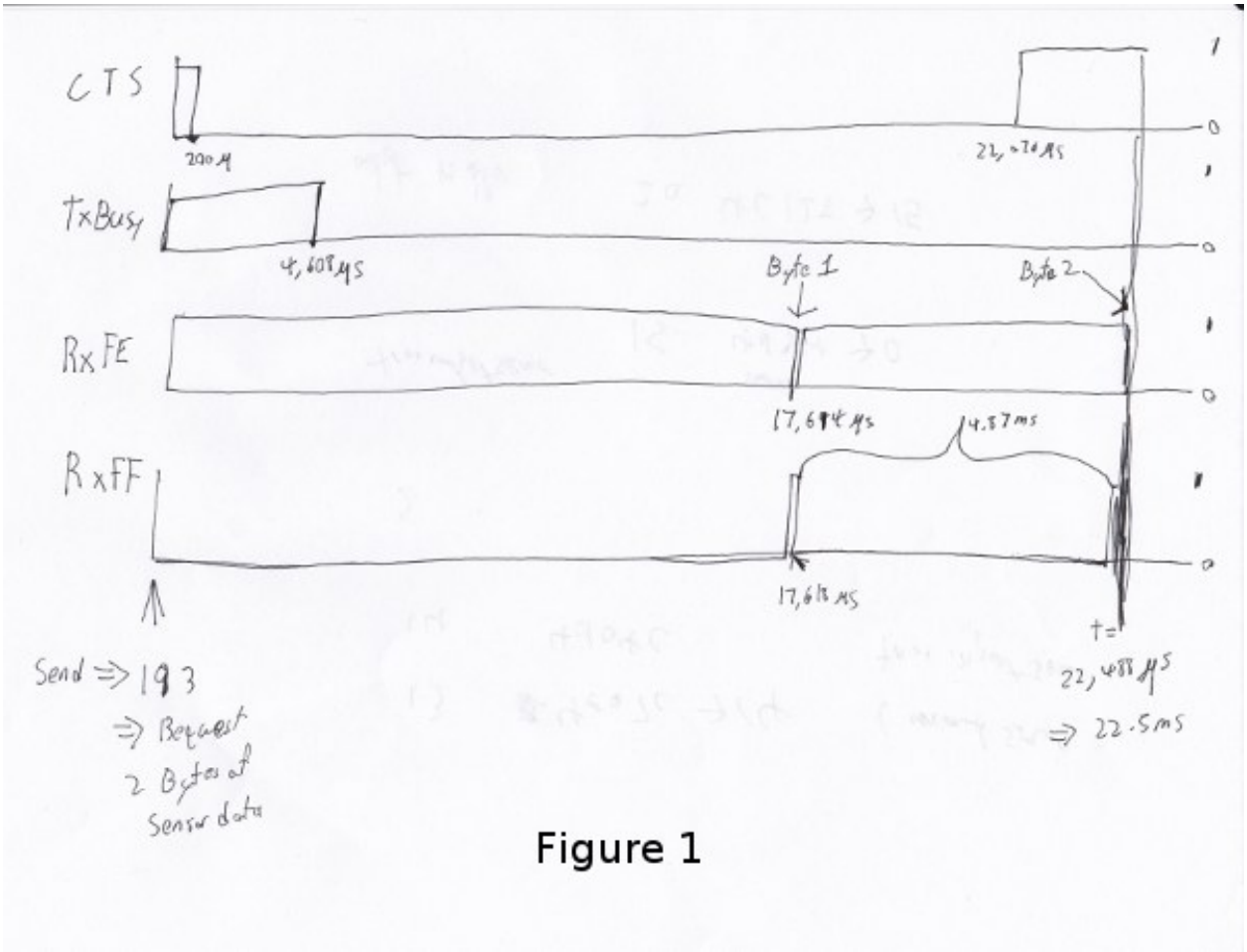

Figure 1

System Calls

- System calls support up to 5 arguments.

`Create` Returns the new task id, `ERR_K_INVALID_PRIORITY -1`, or `ERR_K_OUT_OF_TD -2`

`MyTid` Returns the current task id

`MyParentTid` Returns the parent task id. The parent task id is always returned regardless of the parent's state.

`Pass` (Rescheduling happens as normal in the background.)

`Exit` Task is marked as `ZOMBIE` (and rescheduling happens as normal in the background).

`Send` Sends a message to the given task ID. `-3` code is not implemented.

`Receive` Blocks until a message is received. Returns the size of the message which will be typically `MESSAGE_SIZE 16`

`Reply` Replies a message to the task. On errors `-3 -4`, an assert will fire before returning to aid in debugging.

`RegisterAs` Prepares a `NameServerMessage` structure with a message type of `REGISTER_AS` and sends the message to the Name Server. `0` is always returned because the Task ID is hard-coded and the call should never send to the wrong task.

WhoIs Prepares a `WHO_IS` message type and sends it to the Name Server. As noted in `RegisterAs`, we either return a Task ID or 0 if the task has not been created. However, the task ID returned may be in a zombie state.

AwaitEvent Marks the task as `EVENT_BLOCKED`. The task will be unblocked by the Scheduler. This call always returns 0 and the user task will be responsible for obtaining the data themselves. `AwaitEvent` supports only 1 task per event type.

Time Wraps a `Send` to the Clock Server. It first queries the Name Server for the Clock Server and then sends a `TIME_REQUEST` message. It expects back a `TIME_REPLY` message and returns the time.

Delay Similar to `Time`, it sends a `DELAY_REQUEST` message and expects back a `DELAY_REPLY` message.

DelayUntil Similar to `Time`, it sends a `DELAY_UNTIL_REQUEST` message and expects back a `DELAY_REPLY` message.

TimeSeconds, DelaySeconds, DelayUntilSeconds Same as above but in seconds. It simply converts the ticks into seconds before calling the system calls. These calls are simply for convenience.

Getc Sends a message to either Keyboard Input Server or Train Input Server. It will block until the servers have a character to return.

Putc Sends a message to either Screen Output Server or Train Output Server. The servers will place the character into the server's Char Buffer.

PutString Formats the string and calls `Putc` for every character.

PutcAtomic Like `Putc`, but accepts multiple characters and guarantees the characters are placed into the queue sequentially. This call is useful to ensure that two byte commands are not separated by a single byte command.

SendTrainCommand Sends a message type `TRAIN_COMMAND` to the Train Command Server. The call is for convenience.

PrintMessage Similar to `PrintMessage`, but this sends the string to the UI Print Server to be displayed on the lower half of the screen using a `UI_PRINT_MESSAGE` message type

Memory model

The memory model looks like this:

```
+----------------+ 0x0200_0000
| RedBoot Stack  |
+----------------+ 0x01fd_cfdc Starting value of redboot stack
| Redboot Buffer*|             after box reset
+----------------+ 0x01FD_B09C
| Kernel Stack   |
+----------------+ 0x01FD_B09C - sizeof(KernelState) - 400kb
| IRQ Stack      |             = KernelEnd
+----------------+ KernelEnd - 500kb
| User Stacks    |
|                |
+----------------+ 0x0005_2804 (_EndOfProgram specified in orex.ld)
| Kernel         |
+----------------+ 0x0004_5000 (%{FREEMEMLO} RedBoot alias)
| RedBoot        |
+----------------+ 0x0000_0000
```

${FREEMEMLO}

After consulting the RedBoot documentation, the entry point was moved to `0x00045000` to free up more memory for user stacks. We believe that this new memory location marks the start of safe memory that is not used as a guarantee from redboot and we have not found any reason we cannot move the entry point to this location. This values comes from the a redboot alias %{FREEMEMLO} that can be used when loading the program instead of the literal address.

As well, we are able to have assert checks on stack boundaries. Using the `_EndOfProgram` linker symbol, we can check if a user stack pointer overwrites the kernel. There are checks for each user stack as well.

Stack values and sizes are configurable, and will generally give appropriate assertions if the memory model has conflicts that can cause corruption.

Message Passing

Messages are `structs` that are casted into `char*`. This casting allows us to manipulate messages more easily with type safety rather than dealing with raw `char`. Note we use GCC attribute syntax to word align the character array as the GCC compiler does not realize we are type punning.

Kernel Messages, messages that are copied into the kernel, are now stored into an array, using Dynamic Memory Allocation (see below), instead of using a combination of ring buffers and queues. Refactoring to a simpler solution allows us to reduce the load on our brain while debugging the kernel. See Dynamic Memory Allocation for more information.

The message size is fixed to 16 bytes. Using a fixed value allows for consistency. As well, this low value is meant to reduce the time spent on message copying.

Priority Queue

The Priority Queue uses 32 levels of priority by using 32 Queues.

Note the highest priority is 0 and the lowest priority is 31. Named priority levels are removed as they were no longer used. Explicit values are now required to remove ambiguity.

When retrieving an item, the Priority Queue uses an integer to track which priority level has items. When a bit is 1, it means there is at least one item in the queue. For example, `00110000...` means there is at least one item in priority 2 and 3 queues. The count leading zero instruction is used so that we avoid checking all 32 queues when getting an item. `0` is returned when there is no item.

We have also centralized all of the priorities into one file called `priorities.h` for easy manipulation.

Interrupt Handler

File: `kernel_irq.c`

Vectored interrupts are used.

Timer3 is enabled and counts down from 5080 to give 10ms interrupt intervals. The kernel also sets the CPSR to allow interrupts.

The interrupt handler will call the scheduler to unblock tasks and it also acknowledge Timer3.

UART1RXINTR1, UART1TXINTR1, UART2RXINTR2, UART2TXINTR2, are enabled when there is a Task waiting for it. The IRQ handlers will disable the respective interrupt after it has fired. UART Clocking problems are avoided as our context switch is greater than 50 NOPs.

Watchdog

A watchdog was added to the scheduler. It runs as the lowest priority task. If the watchdog is not scheduled within 1,000,000 rounds, the scheduler will dump out task statistics and hang. This watchdog will indicate if any tasks are starved. If this condition does occur, it will report within a minute.

Assert

The assert statement has been enhanced to show Thomas The Tank Engine. Please do not be alarmed when you see it.

Bugs have been fixed related to entering assertion failure mode and it should not work properly from user mode, supervisor mode, irq mode, and in the presence of premption.

Serial IO

File: `uart.c`

- FIFOs were not used for this deliverable.

The following Serial IO notifiers call `AwaitEvent`

Task	Event ID	Reports to
Keyboard Input Notifier	UART2_RX_-EVENT	Keyboard Input Server
Screen Output Notifier	UART2_TX_-EVENT	Screen Output Server
Train Input Notifier	UART1_RX_-EVENT	Train Input Server
Train Output Notifier	UART1_TX_-EVENT	Train Output Server

UART Bootstrap Task

The UART Bootstrap Task is responsible for setting up the UART clock speeds and settings. It also starts up the servers.

Keyboard Input Server, Train Input Server

The Input Servers receive keyboard and train inputs. They have a Char Buffer and receive byte data as notified. `Getc` callers will have their task IDs queued. Once Char Buffer contains data, the `Getc` callers will be replied with the character.

Screen Output Server, Train Output Server

The Output Servers send screen and train outputs. They have a Char Buffer and send bytes as notified. `Putc` callers will send the character to the server and the character is queued onto the Char Buffer. Once it is OK to transmit by checking the CTS flag, the character is popped from the Char Buffer and transmitted.

Train Servers

File: `train.c`

Train Server

The Train Server is responsible for handling sensor data from the Train Sensor Reader and queries from the UI Server. It also starts the Train Sensor Reader and Train Command Server

Data Structures The Train Server stores its sensor data into bit flags. The least significant bit represents the first sensor. This scheme allows easier masking:

- `flag & 1<<0` is the first sensor

- `flag & 1<<1` is the second sensor
- `flag & 1<<15` is the 16th sensor

As well, the Train Server stores the last Time the sensor was triggered.
The data structure we use for train track navigation is the track graph provided on the course website.

Train Sensor Reader

The Train Sensor Reader task is responsible for sending track sensor commands and reading them from the train controller. It calls the Train Command Server for the data and manipulates the bytes into a easier to handle form. It then sends the values to the Train Server.

Train Command Server

The Train Command Server is responsible for receiving Train Command messages such as `SPEED` and `READ_SENSOR`. It calls `Putc` and `Getc` as required. Passing all train commands through this server is a form of mutual exclusion. It ensures that commands are fully sent to the trains and commands are not mangled by different tasks.

Train Navigation

File: `route.c`, `tracks/track_data.c`

Train navigation is currently accomplished using naive graph search algorithms, as well as a server called the SwitchMaster that is responsible for updating the positions of switches.

We have broken down the problem of navigation to anywhere on the map into two basic problems: The first is navigation to a point while considering the map as a directed graph. In this situation we only consider moving in the forward direction. In this context, it is not possible to navigate to anywhere on the map from all nodes because the graph is considered to be a directed one. In the second case, we consider the map as an undirected graph, where any shortest path can be found by finding the shortest route in the undirected graph. We can then express the problem of navigation between two points in the undirected graph as multiple navigations in a directed graph, while adding direction reversals in the middle.

Stopping

For stopping we use a roughly approximated table that will tell us how many millimeters before a sensor we need to issue a command to slow down. This table was derived from empirical measurements and still needs a bit of calibration. This is especially true on a specific train level, since different trains require different stopping distances.

Velocity

Our trains move at a speed of 50 cm/s and we maintain this speed using a simple feedback control mechanism. This is accomplished by a simple algorithm that increases the train speed when it arrives at a sensor too slowly, and decreases when it arrives too fast.

Sensor Malfunctions

Sensor malfunctions are accounted for by maintaining a list of sensors that are known to malfunction on each track. We use a blacklist of sensors to remember which sensors should not be navigated to, and which should be ignored when determining the train position.

Train Switch Master

The Switch Master is responsible for picking up switch commands from the Train Server and calling Train Command Server. This task is a worker that removes the burden of waiting for train commands to complete.

Train Engine Client

The Engine Client is responsible for picking up train speed commands from the Train Server and calling the Train Command Server. Like the Switch Master, the task is a worker hired by the Train Server.

Train Engine States

Name	Description
IDLE	The engine is stopped and waiting.
FINDING_POSITION	The engine is moving slowly and waiting for a sensor
FOUND_STARTING_POSITION	The engine has found its location and is calculating a path to the destination
RUNNING	The engine is running at high speeds to the destination
AT_DESTINATION	The engine is at the destination and stopped.
NEAR_DESTINATION	The engine has slowed down and is waiting for a sensor report.
REVERSE_AND_TRY_AGAIN	The engine is in a direction that provides no destination and is reversing to find a new sensor.

GO

The go command operates as following:

1. Set the train speed to 4.
2. If a sensor is hit, pick a random destination.
3. Calculate a route to the destination.
4. Speed up the train to 11.
5. Using feedback control system, adjust the speed to achieve a speed of 50 cm/s.
6. If the distance to destination is within the stopping distance, slow the train down.
7. Wait for a sensor and stop.

For an iterative version of the go command, see GF command which will iteratively use the go command after a train reaches its destination.

GF

The gf command operates as following:

1. Do steps 1-7 of the go command
2. goto step 1

UI Servers

Files used by UI servers: `ui.c`, `ansi.c`, `maps/map_gen.py`, `maps/map_a.txt`, `maps/map_b.txt`

UI Server

- Backspace has been fixed.
- Minor bug: certain inputs will cause assertion failures.

The UI Server is responsible for drawing the textual user interface. It draws a header, the time since start up, the command prompt, table of sensors readings, an ASCII diagram of the track layout, train status, and a scrolled area of train information.

The command prompt supports up to 80 characters. Once this limit is reached, no input will be accepted and displayed. It supports backspace. Pressing the Enter key will execute the command and a response will be displayed under the command prompt.

When a sensor is triggered, the UI Server will display an bold number on the table. Sensor data for the UI is cached by the Train Server so displayed sensor readings may not reflect actual state. Sensor states in the Train Server, however, reflect actual states.

The ASCII map shows sensors as X and bold X. Switches are shown as U, C, or S which represent Unknown, Curved, or Straight. The ASCII map code was generated through a script from a text file.

A green highlight shows the destination. Bug: the green highlight is not persistent if an updated sensor overwrites the cell.

Some of the hilights of the UI are found in figure 4.

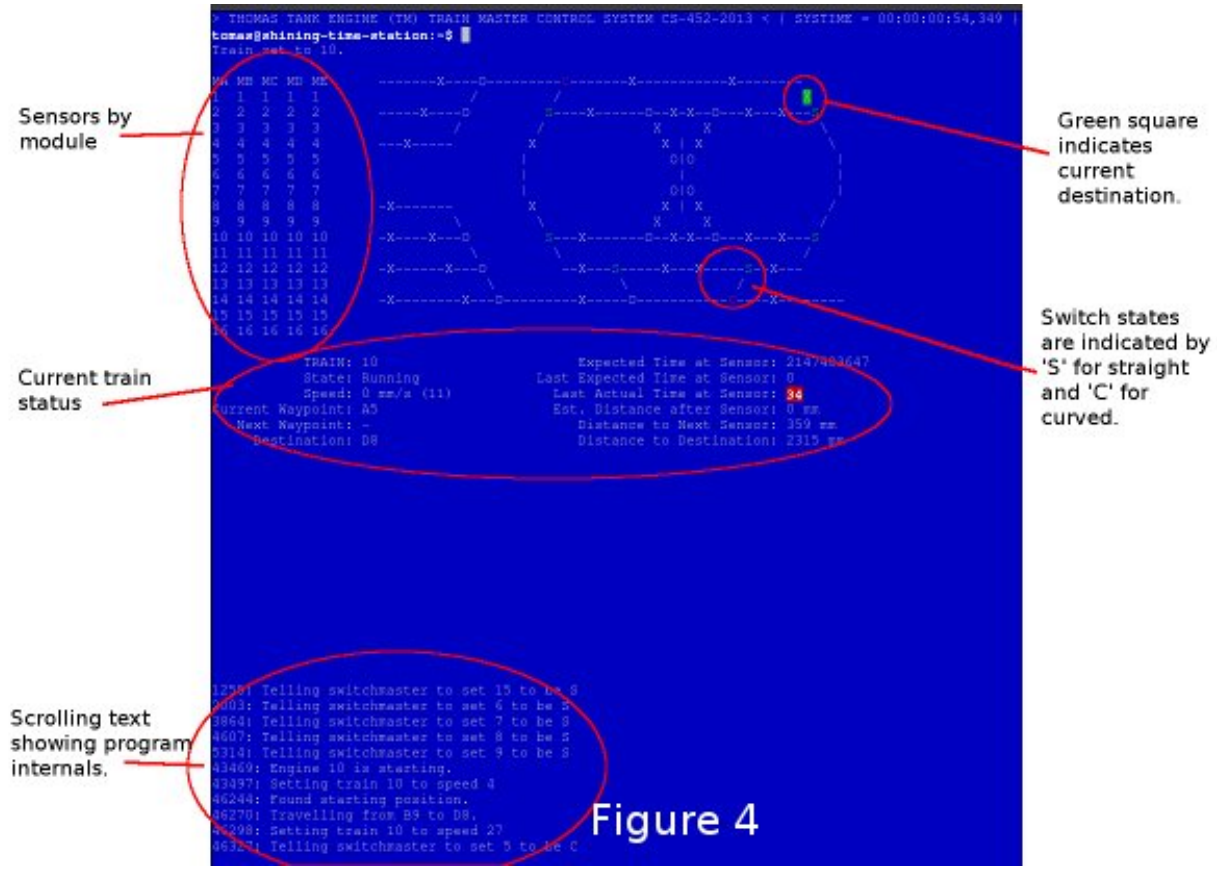

Figure 4

UI Timer

The UI Timer is responsible for sending a message to the UI Server. The timer tells the UI to update the clock on the screen.

UI Keyboard Input Task

The UI Keyboard Input task is responsible for calling `Getc` and sending the character to the UI Server.

UI Print Message Task

This task is responsible for printing messages into the scrolled area. It uses the ANSI feature to set scrolling areas. It is separate from the UI Server as messages may be from higher priority tasks like the Train Server. It is called via the `PrintMessage` call. This method was implemented as a non busy-waiting alternative for debug messages.

Performance

In this deliverable we have several features that significantly improve the performance of our kernel:

1) Time slicing has been reduced to grant each task a maximum of 700 microseconds. This significantly improves responsiveness.

2) Works with all gcc optimization levels.

Source Code

The source code is located at `/u4/chfoo/cs452/group/p1-submit/io/project1/`. It can be compiled by running `make`.

Source code MD5 hashes:

```
chfoo@linux032:~/cs452/group/p1-submit/io/project1$ md5sum */* *.*
50ef0e1e3c71ab1e795fc3d39f75ef9d  include/bwio.h
9af226f127c1fd759530cd45236c37b8  include/ts7200.h
3dfa3ed141445a72c20840b384c1ebb9  maps/map_a.c
c6adb76c95a6ae7986d03cd416d5837e  maps/map_a.h
703f1eeadf245074517591baa0844a37  maps/map_a.txt
1bc708755da7c2295b062a14b0185558  maps/map_b.c
eba8710b29615da70e7165571efd99d8  maps/map_b.h
b1999e3d216a76638d8b74ad993082cf  maps/map_b.txt
ead84e8315fd7e45f0e8e631197b9150  maps/map_gen.py
94944e9febc4db1bb344fff990ed7e9e  maps/map.h
cc1cbe679f12e26e95e6580ca063ebe5  tracks/track_data.c
1352f3743944badbb8c2399e6fb2ccd4  tracks/track_data.h
597a69fd6868b990814a1c3a7dbdd9a3  tracks/track_node.h
0f9e8e1f2726f15e222795960bbbbf8c  ansi.c
cc47d9653ed272a2d23a743ab186914d  ansi.h
b8c8b5fafcd1fd43beaeee7da1e5550f  buffer.c
04c39523dd006155ba353fb3ba1dddfb  buffer.h
ad48b92a01b68f1b8e33f95a9590e7f9  clock.c
f798d08d32ce37146d8013b821f740f5  clock.h
d79855f9ffb6a0003409ebb81290b47f  figure1.jpg
ea9ed6320aea54e698752e9a9b94adc5  figure2.jpg
4bc0f85c30a9d3bfaf7d355123aadf58  figure3.jpg
9adce26681f68a082f5c45bf7833c0ed  figure4.jpg
796800c7dc1bbd2d2444ff3ad2046a51  ioflags.jpg
cac2aaebb371f2ab8150cdbe1e7f5528  kern.c
90a5077c8acbf20134123595fa4189eb  kern.elf
b05992a75764f4239db6b5f47e3a2b75  kernel_irq.c
```

```
db3b8b5c5eaa48d2e5bab408ffd172c2  kernel_irq.h
5313c05e6242631f379b5141ebca4f5f  kernel_state.h
d41d8cd98f00b204e9800998ecf8427e  kern.h
5439df921ac46fd07959e43125fefa91  memory.c
b16265e8b0bfe3a510b3a25e05b8674a  memory.h
adcff2244ac92050360eacd7ab4f5dd9  message.c
d6c6d5d6b12cfa2e26b60c7097190e35  message.h
615b2439e1f227fc8451bce70c045e11  nameserver.c
f9335969b8c71be878a915c26e7a606c  nameserver.h
781c959d1329e2f98aad3b782bea50f4  notifier.c
3fd892b4a7ec6c055cdad49ad7449b59  notifier.h
78a32a3a80cad8a4cc40de1ce18fbe29  orex.ld
ff0c679a0b4d9d358874bd98202942d2  priorities.h
cf633eed1c5eaa9cb54a2f74f1d34fa2  private_kernel_interface.c
837c722d6fe58cb4998e0745bdba768a  private_kernel_interface.h
2a63325e4fb036a5acf37c8bd63ee4ca  public_kernel_interface.c
93963b17c60bdd40b39629a70d43405b  public_kernel_interface.h
63c2ccbe48bb263149cfdc1d0cbe0370  queue.c
c205f1a754b08e3a6a236431e441e419  queue.h
cd7239008bf3fc8474819d9183b0cc0f  random.c
7b31c57ff692317d816c839156382596  random.h
3f36d9a8c9e773e144a3ff73951392c8  readme.pdf
e9fd61693ae1bb3b59b13de9076b87c6  readme.rst
2f81e678122b70367d3a08c645e2c53a  readme.tex
25161c0544bf9211c20b350d3fefcef0  robio.c
41eb1d6816c426b4e085e0d123ceb456  robio.h
03efe383d78a17931a84aeaa3c0473cd  route.c
9e6db63fecbd833ce0b6f29532a8e011  route.h
62ada229addeb4cac6200c1e2f0f5621  rps.c
222f8edffbfde11ee553b7561f4c10a8  rps.h
ffe74790069725c8cd0fc5d1d872a7d0  scheduler.c
aabd27e8d3fb723a02c437df334983fa  scheduler.h
25aa08b383825ef2ce548d255503f275  Screenshot-1.png
e613d497f4ddd240605c62968fcc8b98  Screenshot-2.png
529204cb3fa263a88c9e866fa0faac39  Screenshot.png
fba4eb1fd2006e2d70124be70af02282  swi_kernel_interface.s
00f9f65864243bdd18687e7a849c72a1  task_descriptor.c
34b26bd48a79c0a2572ca700e9ea4283  task_descriptor.h
c534daa815c3638a88cebc7caf8b9d6f  tasks.c
0d3699b1a8224eb6995bb042834f66b5  tasks.h
eaed4bd78a6fa73453c639c426fef6b6  test_uart.c
5b820ca4fce39820f678a6080fd594ef  test_uart.h
8e0691e496fc8e75b861f4fd94641239  train.c
6231509c4982641abf9fab9b2baed473  train.h
f9fed4dbdfd0559f58a03c8b3df8c216  uart.c
1a8185a782b5c582a6ba13127ae1a1e3  uart.h
bb50f6b51470e70c2c2826b0639456dc  ui.c
9a7bc0d6fbcbb469da6352129ba60b29  ui.h
5b609bdd0235c3858e16c053b8e53bfd  va_list_def.h
```

Elf MD5 hash:

```
chfoo@linux032:/u/cs452/tftp/ARM/relder-chfoo/p1-
```

```
submit$ md5sum kern.elf
90a5077c8acbf20134123595fa4189eb  kern.elf
```

Git sha1 hash: `3b087bd471f95d9ebfdb19129507cb1279b45fe7`