CS 452 P2

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

Date due: July 23, 2013

Running

The executable is located at `/u/cs452/tftp/ARM/relder-chfoo/p2-submit/kern.elf`.

The entry point is located at `0x00045000` or `%{FREEMEMLO}` It *must* be executed with caching enabled. (Caches not enabled by the program itself due to time contraints):

```
load -b %{FREEMEMLO} -h 10.15.167.4 ARM/relder-chfoo/p2-
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.

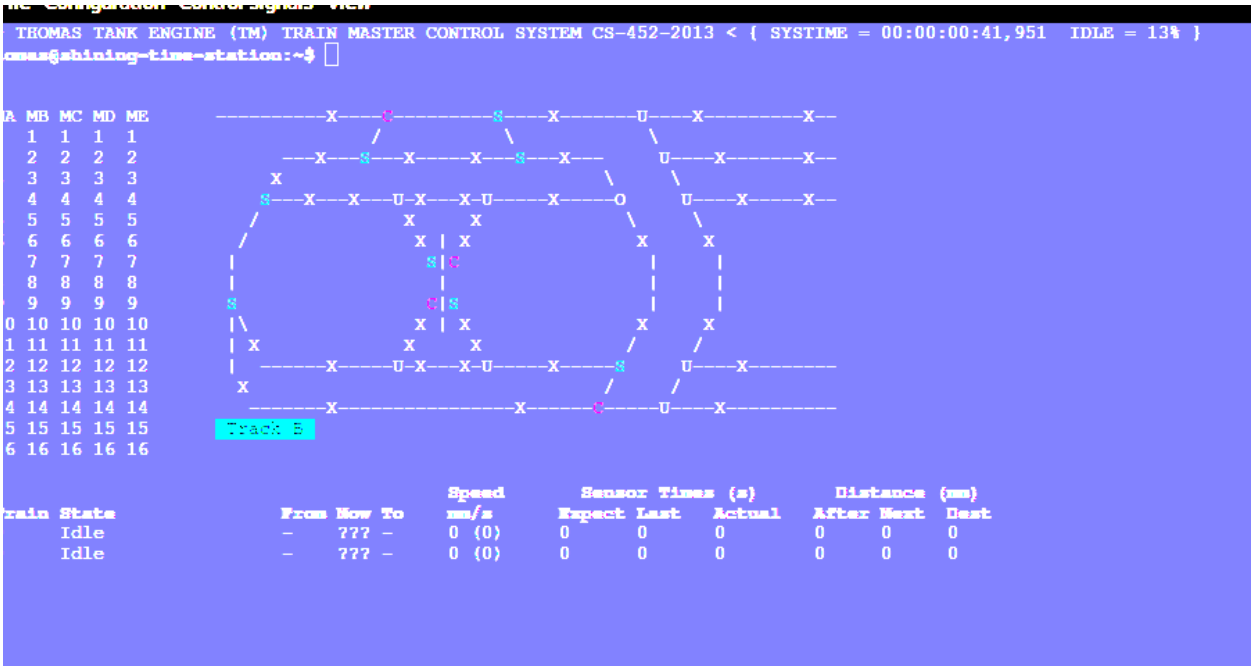

Figure 1: Figure 2

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.

num TRAINS Set the number of trains to be used.

paint Causes the interface to redraw itself.

rt Resets the train system by stopping the trains, clearing reservations, and clearing train engine states.

rps Runs Rock Paper Scissors program.

Pressing 'CTRL+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.

Pressing CTRL+C will cause the program to exit immediately without shutting down the tasks.

Getting Started Quickly

To start up two trains

1. Select the appropriate map using the `map` command.
2. Set the number of trains to be used using `num` command.
3. Enter the first train using `go TRAINNUM1 0`
4. Enter the second train using `go TRAINNUM2 1`
5. If things go wrong, use the `rt` command

Description

Kernel

- No kernel changes since last deliverable.

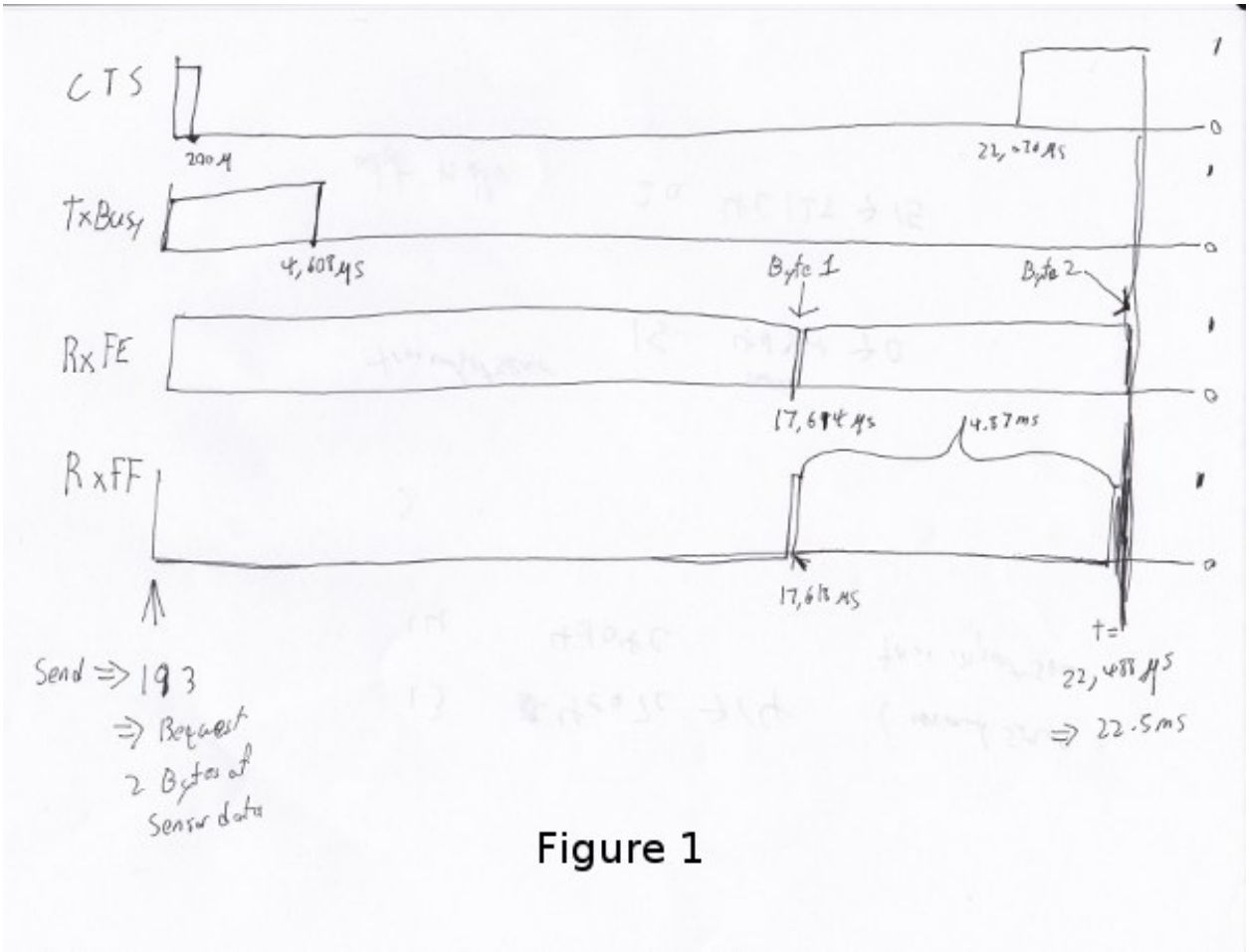

Figure 1

System Calls

- System calls support up to 5 arguments.
- No changes since last deliverable.

`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

Watchdog

The watchdog has been changed to report starvation after 500,000 schedules to be more strict in detecting this problem.

Scheduler

The scheduler now calculates the system load by counting the number of low priority schedules per 1,000,000 schedules. This may not reflect the true load as the Idle Task may take a long time slice before rescheduling. In the future deliverable, we may implement counting the time each task is scheduled.

Priorities

For this deliverable, we have thought carefully about the priorities of each task.

Task	Priority
Clock Notifier	0
Clock Server	0
First Task	0
Name Server	1
Administrator	2
UART Bootstrap	3
Train IO Notifier	4
Train Input Notifier	4
Train Output Notifier	4
Keyboard Input Notifier	4
Screen Output Notifier	4
Train Input Server	5
Train Output Server	5
Screen Output Server	6
Keyboard Input Server	6
Train Server	7
UI Print Task	7
Train Command Server	8
Train Switch Master	8
UI Server	8
Train Sensor Reader	9
Train Engine	9
Train Server Timer	10
UI Keyboard Input	12
UI Timer	13
RPS Test Start	15
RPS Server	16
RPS Client	31
Idle Task	31

For more info, see Performance.

Assert

The assert statement, as usual, is enhanced to show Thomas The Tank Engine. Please do not be alarmed when you see it.

When an assertion failure occurs, the Stop command will now be sent to avoid train collisions.

Serial IO

File: `uart.c`

- FIFOs are now used for the terminal input/output.

Train Navigation

File: `route.c`, `tracks/track_data.c`, `train_logic.c`, `train_data_structures.h`

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.

To find a destination, a simple depth first recursive algorithm is used to build up a Route Info array. The Route Info array contains information about each track node and the switches it needs to switch. The algorithm avoids blacklisted switches.

Undirected Graph Model

In order to accurately model the train and its motion around the track, as well as predicted future positions on the track, we required another representation of the track to complemented the directed model that was provided. It is for this reason that we have created a undirected graph model of the track based on the directed graph model. This model also includes the trains as nodes, which enables us to apply standard graph-based algorithms to any nodes on the track graph, including the trains themselves. This has significant advantages for tasks such as sensor attribution, collision detection, and route planning. The advantage of including the trains as nodes in this model means that in this representation, we do not need sensor data to make decisions about what actions to take, and can rely on the current state of the model that has been predicted based on last sensor observations. The undirected graph model allows us to consider route planning, independent of the number of reversals that are required on the route. The other advantage is that trains are included as nodes so that the shortest distance between two trains can be calculated down to the micrometer at any point in time, as long as their approximate speed is known.

Sensor triggering can be used to infer observed train speeds, which can be used to simulate the motion of the train in a near continuous time manner.

Undirected Graph Data Structure

The undirected graph model is built from the directed track node data. Pointers are added to the directed nodes that point to the corresponding undirected graph nodes, and vice versa. The undirected graph model is implemented as an adjacency list. Since every node in this graph can have a maximum of 3 adjacent nodes, this significantly shortens the run time and memory requirements of many graph processing algorithms.

Dijkstra's

Dijkstra's algorithm has been implemented for the undirected graph nodes. The implementation of this algorithm is the standard one, with a run-time of $O(|E|+|V|)$. Testing has been done with a simulated track where multiple trains are sent on a random-walk around the track millions of times, calculating the shortest distance at each step. Valgrind was also used to preclude the possibility of programming errors.

Routing and Navigation

Currently, we use a simple recursive graph search algorithm for calculating paths. This will soon be replaced by the much more accurate Dijkstra's algorithm once the undirected graph model is incorporated into the routing. Once we have determined a series of nodes that we need to navigate through, we determine the set of switches that need to be changed from their current state, up until we possibly end up changing that same switch again (for re-entrant paths

that only involve moving forward). The switches are queued in the order in which they need to be switched so that the closest switch will be the first one to change. If the train triggers a sensor that is not on the path it was expected to take, a warning is printed for debugging purposes.

Stopping

For stopping we use a roughly approximated table for each train 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.

A list of speeds for each node during stopping has also been determined empirically. Nodes that are near switches have a lower speed to avoid stopping on top of a switch. We risk the trains getting stuck on curves because it is preferred that trains become stuck rather than derailed by an activating switch.

Velocity

Our trains move at a speed of 45 cm/s and we maintain this speed using a feedback control mechanism. The observed train speed is calculated by dividing the known track length between two sensors, and dividing this by the observed time taken to travel between them. The trains use a floating point speed setting to avoid sending too many train speed commands and to dampen noise. The floating point speed setting is casted to an int and the command is issued if needed. The algorithm slowly increases the train speed when it arrives at a sensor too slowly, and decreases the speed quickly 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.

Reservations

The provided track nodes have been modified with an extra field called `reserved`. It holds the train number of the reservation. Once the destination and route is calculated, all the nodes in the route are reserved. Once the train reaches its destination, the nodes are released from reservation. The concept of switch reservations is taken care of, because while a train has reserved a switch, no other can attempt to queue a switch change.

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

... continued on next page

Name	Description
RESYNC_POSITION	The engine has drifted from its calculated position and is attempting to find its location
FOUND_STARTING_POSITION	The engine has found its location
WAIT_FOR_DESTINATION	The engine is waiting for a destination to be calculated
GOT_DESTINATION	The engine has calculated its destination
WAIT_FOR_ALL_READY	The engine is waiting for other engines to be found and ready
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.
WAIT_FOR_RESERVATION	The engine has stopped and is waiting for the track to become unreserved
WRONG_LOCATION	The engine has entered an unauthorized section of the track

GO

The go command operates as following:

1. Set the train speed to 5.
2. If a sensor is hit, the location of the train has been found.
3. Reserve the current location in the reservation system.
4. If there are other trains that need to find their location, wait for them.
5. Pick a random destination.
6. Calculate a route to the destination.
7. If there is no possible route to destination, reverse the direction and go to step 5.
8. Activate the switches that do not overlap other routes or require switching multiple times.
9. Speed up the train to 14.
10. Read sensors and compute the speed, location, and distance to update the state of the train engine.
11. Using the sensor data and feedback control system, adjust the speed to achieve a speed of 45 cm/s.
12. If the next node is a switch that needs to be activated, switch it.
13. If the distance to destination is within the stopping distance, slow the train down.
14. 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 1-14 of the go command
2. Wait for 4 seconds
3. 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

- CR LF is handled correctly now.
- 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, a system load indicator expressed in percentage, the command prompt, table of sensors readings, an ASCII diagram of the track layout, a table of 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. If an error occurs, it will be shown in yellow.

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 of the first train. A yellow highlight shows the destination for other trains.

A black highlight shows the reservation of the first train. A red highlight shows the reservation for other trains.

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

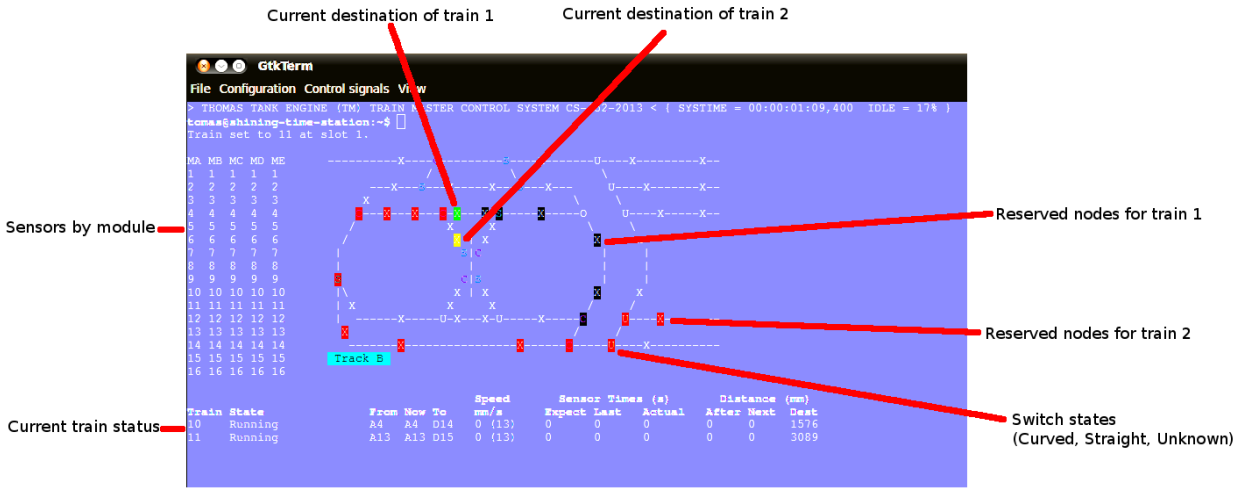

Figure 2: Figure 4

UI Timer

The UI Timer is responsible for sending a message to the UI Server. The timer tells the UI to update the clock and system load 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.

Rock Paper Scissors

Rock Paper Scissors is now back and can be run using the `rps` command. Since it still functions from our earlier deliverable, we have decided to use it for stress testing. It currently runs with 42 clients players. It will display the results of each round in the scrolling area of the UI. The `rps` command should only be run once, however, the RPS games will last 12345689 rounds so there is no need to rerun the `rps` command the second time.

Performance

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

1) The priorities were adjusted to achieve the following
   - Notifiers have high priority
   - The UI keyboard input no longer drops characters while the UI is redrawing.
   - The Switch Master and Train Speed Client are at higher priorities than the Sensor Reader. This setup is necessary to avoid the trains getting caught on the switches.
2) FIFOs for the terminal were enabled. Without FIFOs, the UI task may be interrupted during sending ANSI sequences and leaving incomplete sequences on the screen. With FIFOs, the screen updates correctly without flickering.

Source Code

The source code is located at `/u4/chfoo/cs452/group/p2-submit/io/project2/`. It can be compiled by running `make`.

Source code MD5 hashes:

```
chfoo@nettop36:~/cs452/group/p2-
submit/io/project2$ md5sum */*/* */*.*  *.*
bd0a0df5b9fbc588bdc203efe3c6570d  tracks/tests/Makefile
fac587c527e77d0b2b243879ce9cc9a3  tracks/tests/tests.c
50ef0e1e3c71ab1e795fc3d39f75ef9d  include/bwio.h
9af226f127c1fd759530cd45236c37b8  include/ts7200.h
94944e9febc4db1bb344fff990ed7e9e  maps/map.h
3dfa3ed141445a72c20840b384c1ebb9  maps/map_a.c
c6adb76c95a6ae7986d03cd416d5837e  maps/map_a.h
703f1eeadf245074517591baa0844a37  maps/map_a.txt
c415ff53472ff6ffcd28afcab0038e4c  maps/map_b.c
eba8710b29615da70e7165571efd99d8  maps/map_b.h
093b6fff35ab1def4b776eb25a623c01  maps/map_b.txt
ead84e8315fd7e45f0e8e631197b9150  maps/map_gen.py
1a1aac0b745639b84fe74f1839547512  tracks/track_data.c
```

```
1352f3743944badbb8c2399e6fb2ccd4  tracks/track_data.h
e33dcce364a34b75f722eb3d272626cb  tracks/track_node.h
c531fbaa2b102637bf455e0f6176bc36  tracks/undirected_nodes.c
1828da1574bab787f726066f173a699d  tracks/undirected_nodes.h
1a5d522885e2e71cd9b940bd52ff9b42  Screenshot-1.png
e613d497f4ddd240605c62968fcc8b98  Screenshot-2.png
e92b7c25883384cd034329580bdb0e5d  Screenshot.png
0dc64506433fa8e40520a29acdae7984  ansi.c
cc47d9653ed272a2d23a743ab186914d  ansi.h
b8c8b5fafcd1fd43beaeee7da1e5550f  buffer.c
04c39523dd006155ba353fb3ba1dddfb  buffer.h
ad48b92a01b68f1b8e33f95a9590e7f9  clock.c
f798d08d32ce37146d8013b821f740f5  clock.h
d79855f9ffb6a0003409ebb81290b47f  figure1.jpg
ea9ed6320aea54e698752e9a9b94adc5  figure2.jpg
97543aad843c35a031e79c5faf4ca957  figure2.png
4bc0f85c30a9d3bfaf7d355123aadf58  figure3.jpg
9adce26681f68a082f5c45bf7833c0ed  figure4.jpg
8c879f7e1e375bc7199895c9ef74d8e3  figure4.png
796800c7dc1bbd2d2444ff3ad2046a51  ioflags.jpg
cac2aaebb371f2ab8150cdbe1e7f5528  kern.c
e3d60e6c74c202e9f5d27c1f80ea4e13  kern.elf
d41d8cd98f00b204e9800998ecf8427e  kern.h
6152637f1334fd74e0eb806912affc59  kernel_irq.c
db3b8b5c5eaa48d2e5bab408ffd172c2  kernel_irq.h
bd3f47ad7601caa6f6a64dbbd77ae784  kernel_state.h
5439df921ac46fd07959e43125fefa91  memory.c
b16265e8b0bfe3a510b3a25e05b8674a  memory.h
adcff2244ac92050360eacd7ab4f5dd9  message.c
921f82dee0e89dd011e179f67d706d03  message.h
615b2439e1f227fc8451bce70c045e11  nameserver.c
f9335969b8c71be878a915c26e7a606c  nameserver.h
08703117df738f05b4ba289925ee7bf9  notifier.c
3fd892b4a7ec6c055cdad49ad7449b59  notifier.h
78a32a3a80cad8a4cc40de1ce18fbe29  orex.ld
96282407319e88eb23bb90a8daf06b9f  priorities.h
cf633eed1c5eaa9cb54a2f74f1d34fa2  private_kernel_interface.c
299821b9f1a7a97ec90a3b8863f67045  private_kernel_interface.h
797daf216f6d2955ab3317c1b94f4648  public_kernel_interface.c
93963b17c60bdd40b39629a70d43405b  public_kernel_interface.h
63c2ccbe48bb263149cfdc1d0cbe0370  queue.c
ca12973f6a47637476903ba771206956  queue.h
092ccec4bf20645fcde14470e074e8ea  random.c
7b31c57ff692317d816c839156382596  random.h
4dc002ebccd6a049d9600e703f53eace  readme.pdf
d5216a1e9b9a6da93135aa089ce58373  readme.rst
335f48d9eb92e9ee949b394c96f05b4f  readme.tex
b58af196bdaf29ff72ff1c20b9e92cca  robio.c
5763b2a44810b6d0afafc27fb88cc7de  robio.h
28986f2c4e92d601a1cb6f1bbd846c7e  route.c
2fe7d2acdae03abb1904c8a460f4d53c  route.h
```

```
155b6b3d1816287618cc197aec5d5884  rps.c
6eee23bcabb82e39ca885de1563eca4f  rps.h
02566388717be1765b35028f7f16bf39  scheduler.c
0b1101123bcff9dbbf9d39542c35aacb  scheduler.h
fba4eb1fd2006e2d70124be70af02282  swi_kernel_interface.s
00f9f65864243bdd18687e7a849c72a1  task_descriptor.c
34b26bd48a79c0a2572ca700e9ea4283  task_descriptor.h
33f883f692ee8388a7f1be0b1409c73f  tasks.c
0d3699b1a8224eb6995bb042834f66b5  tasks.h
eaed4bd78a6fa73453c639c426fef6b6  test_uart.c
5b820ca4fce39820f678a6080fd594ef  test_uart.h
b1f93e60c825a24ec38e015fe28d0295  train.c
1c3cc69bfc281b1994fb9ce3d03c129b  train.h
3a70433035aa2ad078efd5f1571eba1a  train_data_structures.h
3a0ed813a12c962e5f741eaacccda1be  train_logic.c
b5cb4a8dc956cc0f45878b0e8b935ad2  train_logic.h
d4ad03272947d96db7fbd9528ca11ced  uart.c
1a8185a782b5c582a6ba13127ae1a1e3  uart.h
833fae6a88b54a60fbde799bab941ca9  ui.c
6fcbf108793dcbca79d022b950830c02  ui.h
5b609bdd0235c3858e16c053b8e53bfd  va_list_def.h
```

Elf MD5 hash:

```
chfoo@nettop36:/u/cs452/tftp/ARM/relder-chfoo/p2-
submit$ md5sum kern.elf
e3d60e6c74c202e9f5d27c1f80ea4e13  kern.elf
```

Git sha1 hash: `1d430103399c522c572e9e0bd71d26ee0d042e51`