Train Control Demo 2 / Final Train Demo

CS452 - Spring 2014

Real-Time Programming

Team (Prepared By)
Max Chen - mqchen
mqchen@uwaterloo.ca

Ford Peprah - hkpeprah
ford.peprah@uwaterloo.ca

Prepared For
Bill Cowan
University of Waterloo

Table of Contents

1 Program Description

1.1 Getting the Program

To run the program, one must have read/write access to the source code, as well as the ability to make and run the program. Before attempting to run the pogram ensure that the following three conditions are met:

- You are currently logged in as one of `cs452`, `mqchen`, or `hkpeprah`.
- You have a directory in which to store the source code, e.g. `~/cs452_microkern_mqchen_hkpeprah`.
- You have a folder on the FTP server with your username, e.g. `/u/cs452/tftp/ARM/cs452`.

First, you must get a copy of the code. To to this, log into one of the aforementioned accounts and change directories to the directory you created above (using `cd`), then run one of

```
git clone file:////u8/hkpeprah/cs452-microkern -b final .
                        or
git clone file:////u7/mqchen/cs452/cs452-microkern -b final .
```

You will now have a working instance of our TC2/Final source code in your current directory. To make the application and upload it to the FTP server at the location listed above (`/u/cs452/tftp/ARM/YOUR_USERNAME`), run `make upload`. This will generate our source code that is functional on Track A, since distances are different on both tracks, you should run by specifying the track you want to use. To do so, you can use any of the options listed below to customize your generated ELF file:

Make Options - Can only specify one

Option	Description
upload	Make and upload the generated file.
debug	Make and upload the generated file with debugging enabled.
test	Make for testing.

Make Flags - Multiple can be specified

Flag	Description
`TRACK=a` or `TRACK=b`	Makes for track A or track B depending on specified.
`SILENT`	Make with debugging off for tests.
`PROFILING`	Make for testing profiling.
`TEST=filename.c`	Make a test file using the specified file as the main.
`TARGET=filename`	Make the code and store it in the specified elf file.

1.2 Running the Program

To run the application, you need to load it into the RedBoot terminal. Ensure you've followed the steps listed above in the "Getting the Program" settings to ensure you have the correct directories and account set up. Navigate to the directory in which you cloned the source code and run `make upload`. The uploaded code should now be located at (depending on the track you made for, defaults to 'a'):

`/u/cs452/tftp/ARM/YOUR_USERNAME/kernel-a.elf` or
`/u/cs452/tftp/ARM/YOUR_USERNAME/kernel-b.elf`

To run the application, go to the RedBoot terminal and run the command

```
load -b 0x00218000 -h 10.15.167.4 ‘‘ARM/YOUR_USERNAME/kernel-a.elf’’; go
```

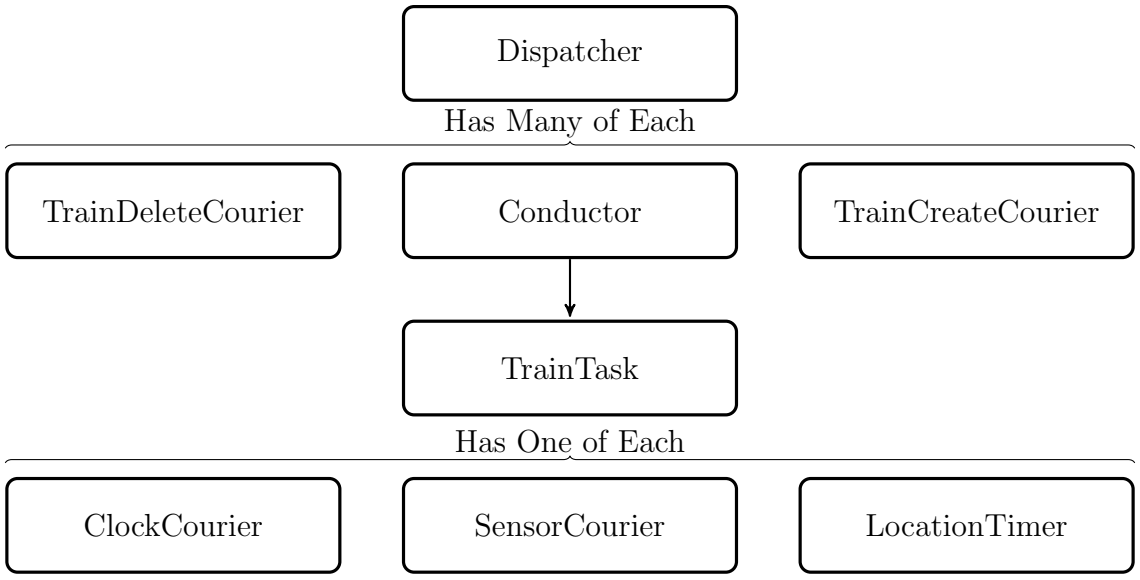
The application should now begin by running through the game tasks before reaching a prompt. The generated files will be located in `DIR/build` where `DIR` is the directory you created in the earlier steps. To access and download an existing version of the code, those can be found at `/u/cs452/tftp/ARM/mqchen/kernel-a.elf` (`kernel-b.elf` for track B), and `/u/cs452/tftp/ARM/hkpeprah/kernel-a.elf` (`kernel-b.elf` for track B).

1.2.1 Command Prompt

After the startup tasks have finished running, the user will reach a command prompt where they will be able to enter commands. A list of available commands and the syntax can be found at run-time by entering either "?" or "help" followed by the "RETURN" key. All commands must be followed by the "RETURN" key for the `Shell` to interpret them.

2 Task Structure of Trains

There are three main tasks used to control the trains: the `Dispatcher`, the `Conductor`, and the `TrainTask`. Aside from them, there are several auxiliary couriers to perform tasks such as waiting on a sensor, waiting for a timeout, sensor attribution, and train deletion. The following diagram outlines the structure of those tasks:



2.1 Couriers

With a robust create/destroy system, all of our couriers are single use, created and destroyed when needed. This keeps them very simple and allows them to exist with almost no internal state. Furthermore, most communication done with couriers are done with no message copies since the context of which courier sent the message is sufficient to determine what event has occurred. In particular, the couriers that follow this methodolgy are the `shortmvDone` and `shortmvTimeout` couriers which are used to signal to issue the sotp command in the short move and that the short move should be completed by now, respectively. As well, the `TrainCreateCourier` and `TrainDeleteCourier` which create and delete trains respectively; these operations are done in couriers to ensure the `Dispatcher` does not block waiting on another task. Lastly, various timeout couriers that simply call `Delay` then respond to their parent task.

2.2 SensorCourier

A courier created by the `TrainTask` to wait on behalf of sensor trips with the `SensorServer`. When the awaiting sensor is tripped, it sends a message back to its parent to notify that the sensor has been tripped.

2.3 LocationTimer

This child task calls into the `TrainTask` once every five ticks to tell the `TrainTask` to update its state and determine if it needs to take any additional actions such as stop, flip switches, reserve more track, and report routing status.

2.4 ClockCouriers

The `ClockServer` has a function that provides a generic courier-based Delay call. This allows for the TrainTask to call Delay without blocking itself and get the response through its Receive. The TrainTask uses the `ClockCouriers` to time its short moves.

2.5 TrainTask

The purpose of the `TrainTask` is

2.6 Dispatcher

The purpose of the `Dispatcher` is to act as a server that mediates interaction to trains between the Transit System (`MrBonesWildRide`), the `Conductor` task(s), and the `TrainTask`(s). In addition, the `Dispatcher` is responsible for the track, which it handles as a resource which tasks may request a segment of to use or release a segment they currently own. To prevent the user from interfering with a routing in process, the `Dispatcher` marks trains that are being routed as `TRAIN_BUSY`; if the user wishes to use this trian, they must then stop the train's routing by executing the terminal command "st TRAIN_NUMBER". All requests made by the user to trains are processed through the `Dispatcher` which then passes them on to the `TrainTask` if is not busy. When a train is added to the system, the `Dispatcher` creates a `TrainCreateCourier` which goes and blocks in creating the train and doing sensor attribution, then calls back into the `Dispatcher` when done; creation is serialized, so multiple trains may be added at once. Sensor attribution works by looking for a sensor that is being triggered (non-faulty trigger), which no other train currently known by the `Dispatcher` is waiting on. When a "goto" command is issued, the `Dispatcher` creates a `Conductor` to find and then execute the path. When a `TrainTask` wants to move the train over a particular segment of track, it must first request that peice of track. This is done through two fo the `Dispatcher`'s calls: `DispatchReserveTrack` and `DispatchReserveTrackDist`, which reserve nodes and nodes by distance respectively. When done with that segment of track, a `TrainTask` must call `DispatchReleaseTrack`. This allows reservation and releasing to be

atomic to avoid race conditions where two trains attempt to reserve the same peice of track. The `Dispatcher` also acts as a lookup servers to find the task identifier of `TrainTask` by the train number.

2.7 Conductor

The purpose of the `Conductor` task is to drive the train from an arbitrary starting point to some distance past a destination sensor. Because some paths involve reversing, short moves, and other intricate maneuvers, the `Conductor` allows from some burden to be relieved from the `TrainTask` by having the `TrainTask` be a stand-alone task responsible only for being aware of its location, reversing the physical train, speeding up the physical train, stopping the physical train, and reserving the nodes nears its location based on where it is moving. A `Conductor` is created when a request is made for a train to travel to a location. The `Conductor` works by first finding a path from the train's current location to the destination node. It then breaks up this path into segments where the transition from one segment to another requires the train to stop and then reverse. It then calls into the train to tell it to move. If the train reports itself as being lost, it calls into the `Conductor` to perform `SensorAttribution` to find out where the train now is. If the train fails to make a reservation, it also calls into the `Conductor` which sleeps then attempts to re-route the train. Once the conductor has successfully routed the train to the specified destination, it broadcasts its arrival, which may be picked up by the Transit System if it exists.

3 Train Control Demo 2

For the purpose of routing our trains from its current location to any provided location, we use a layered architecture in which each feature is built and abstracted such that more advanced navigation systems can be created on top of them without worrying about how they work. We provide an extremely simple interface (include/train_task.h) to an extremely complicated task (src/train/train_task.c).

3.1 Basic Train Driving

The most basic functionality of the train is to just drive and know where it is (required for Demo 1). We accomplish this by having each `TrainTask` create a slave task that sends into it every 5 ticks telling it to update its location, as well as a hard resync with the track whenever

our train hits a sensor. The train tracks a huge amount of information, see Appendix A for details.
Internal to the `TrainTask`, these are provided by the functions `setTrainSpeed` (change speed) and `trainDir` (reverse). These are exposed as functions (wrappers for Send to the `TrainTask`) `TrSpeed` and `TrDir` respectively.

3.2 Path Finding

Path finding is implemented over the provided track graph using the heap version of Dijkstra's, taking into account track reservation as well. If any node is reserved by another train, then the pathfinding algorithm will no longer use that node. For each node, the algorithm will add its reverse, its straight edge, and in the case of branches, its curved edge. The function `findPath` will fill the provided `track_node*` array with nodes of the path, return the number of nodes and write the total length of the path in mm in an output parameter.

3.3 Reservation

The reservation system was built into the provided `track_node` graph by simply adding a field in the `track_node` struct called `reservedBy` that indicates the train number of the train that has the particular node reserved. This could have been done in a more encapsulated way that hides this reservation as the current method allows anyone with a `track_node*` to modify the value. However, we decided to take the simpler approach as time was limited, and trust that our two programmers will not modify the value directly, and instead use the provided interfaces.
At the heart of the reservation system is a compare-and-swap (CAS) function that isn't absolutely true to its name. The function does the simple operation of comparing the existing reservedBy field to a provided value, and if they match, sets it to the new value (more of a test-and-set). The operation is also idempotent, so when a train tries to reserve a node it already owns the operation will succeed even though the comparison failed. The train number -1 is defined / used as RESERVED_BY_NOBODY to indicate that a node is free. As such, a reserve and release simply call CAS for each provided `track_node*` from nobody to the train or the train to nobody, respectively.
When a train needs to reserve more track, it should not modify `reservedBy`. Instead, it calls into a server, the `Dispatcher`), so that modifications to the field are mutually exclusive. The functions (as always, wrappers for a Send into the task) provide a train number and an array of `track_node*`, and optionally a distance. The reservation system will then attempt to reserve track, following the array until it is exhausted. If a distance is provided, then the system will reserve at most the provided distance, no more and no less. That is, if the provided path array is longer than the distance, only the distance will be reserved.

Conversely, if the provided path is not long enough for the distance, then the reservation system will look ahead in the track graph and reserve until the provided distance is met (one minor problem with this is that since we only have `track_node` granularity, we are prone to over-reserving a large portion of the track when we reach an extremely long edge).
The reservation code will return the number of `track_node` successfully reserved, as well as modify the provided distance field (passed in as a pointer) such that it becomes the amount of distance that was not reserved.

3.4 Collision Avoidance

When a train asks for more track, it always provides a distance which is slightly greater than its stopping distance by some factor. This ensure that the train will be able to stop in the amount of track it has reserved. If the stopping distance field is ever positive (that is, the reservation system could not reserve all the provided distance) then the train knows that it is approaching track reserved by another train, and will stop itself. Otherwise, the train obtains a value that indicates how much extra distance it has before it must reserve more track. Once that extra distance runs out, it will attempt to reserve track again.

3.5 Basic Path Execution

Basic path execution is done solely within the train, and does not support paths that include reversals. The function `TrGotoAfter` is provided by the train that will drive the train, in a linear fashion, to some offset after the provided `track_node`. This is accomplished by recomputing the path length on every train state update and then issuing the train speed command when the stopping distance is equal to the remaining path length.
Path execution is done every time new nodes are reserved. When a node is reserved and the previous node is a switch, then the switch is flipped to the state such that moving over the switch will lead us to the correct next node.

3.6 Short Moves

Short moves are accomplished by setting the train to a speed, waiting for some number of ticks, and then stopping the train. They exist almost entirely outside of the normal train operations, and the instantaneous speed of the train is not accounted for during a short move. Instead, we empirically measured the amount of time required to wait to move certain distances, and built a function $f(x) -> t$ which maps distance to travel (x) into time to wait before sending the stop command (t).

When a train is sent a `TrGotoAfter` command, it evaluates the length of the path. If the path length is under the threshold for shortmoves, then it will initiate a short move by setting its speed to our calibrated short move speed, creating 2 time couriers with values $f(x)$ and $2f(x)$, shortMvStop and shortMvDone. Stop is when the train stop command is issued, and done is when the train is finished moving. During the short move, sensors will still sync the train's position but nothing else will.

3.7 Advanced Path Execution

Advanced path execution is accomplished through a task external to the train, the `Conductor`. An instance of this task is created when the `Dispatcher` is asked to route a train to a particular location. The `Conductor` will find the total path, then break it down to path segments that do not require reversing. It will call `TrGotoAfter` with an offset (roughly 300 mm) for each partial path. `TrGotoAfter` is a blocking call that will return when the train has finished its path. After each partial path has completed, the conductor will reverse the train and execute the next partial path.
Since partial paths are overshooting, the train corrects for this internally. This is done by doing another short `findPath` call inside the train when it is initially given a path. If the current train edge is not connected to the start of the path, then a path between the edge and the start of the path will be found internally and then reserved and executed.

3.8 Recovery

Recovery is done in the `Conductor`. Each `TrGotoAfter` returns a `GotoResult_t` which indicates how the train has executed the path, which can be GOTO_COMPLETE, GOTO_REROUTE or GOTO_LOST.In the first case, the conductor will execute the next partial path. REROUTE occurs when the train cannot finish its route but still knows where it is, so the conductor will call findPath again to generate a new path for the train. LOST occurs when the train no longer knows its location. In this case, the train will be moved slowly until it hits a sensor and finds its location again.
There is an issue with the LOST system in that it is hacked together using the train initialization sequence by destroying the train task and adding it again instead of internally to the train. Because of this, the train loses its reservations during the LOST stage and is prone to collisions. We wanted to implement the LOST error recovery within the train but did not have enough time.

4 Final Demo - Transit System

4.1 Overview

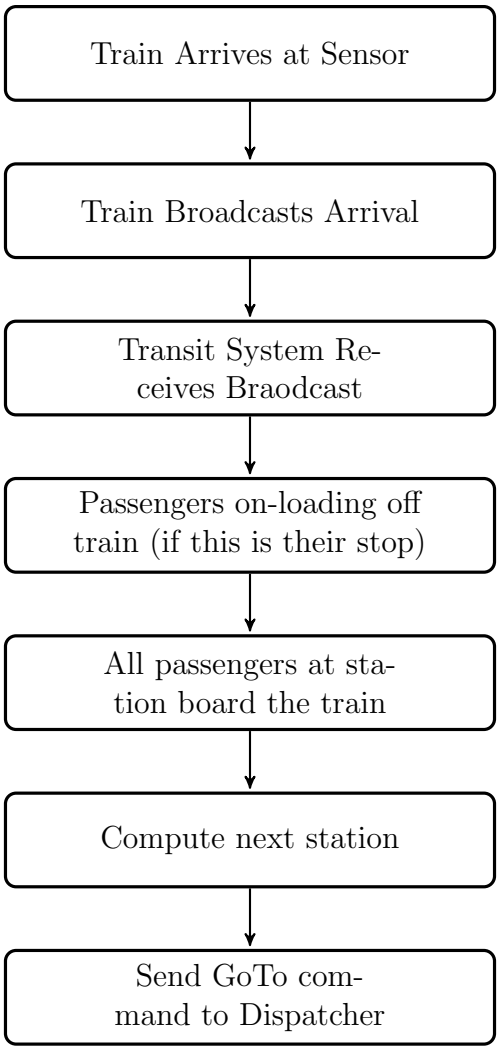
The final demo consists of modeling a transit system through the use of the kernel and code completed from Train Demo 1 and 2. By using path finding, the trains are able to be routed to different stations (represented by sensors) to pick up passengers and bring them to their destination. Each passenger has their own destination of one of the available stations; multiple passengers waiting at one station may each have a different destination. Passengers and stations can be added at will or randomly as the user seems fit. The transit system used in our project has been named "Mr. Bone's Wild Ride". The transit system has two modes; free-standing and monitored. In monitored mode, the user adds the stations by entering the command "spawn", then can add passengers at will. In free-standing, the transit system adds passengers on demand, this is done by the user entering "spawn" followed by "pool". Regardless of the system chosen, to indicate a train is available to the system, the user must enter the command "broadcast TRAIN_NUMBER SENSOR" to alert the system to a new train. A sample session is shown below:

```
mqchen@rtfolks $ spawn 8
mqchen@rtfolks $ pool
mqchen@rtfolks $ add 56
mqchen@rtfolks $ add 45
mqchen@rtfolks $ broadcast 56 A5
mqchen@rtfolks $ broadcast 45 A15
```

This would generate add eight stations to the transit system, then trains 56 and 45, and then alert the transit system that train 56 is at sensor $A5$, and train 45 at sensor $A15$. The `pool` command would cause trains and passengers to be added randomly after a variable amount of time has passed.

4.2 Structure

The following diagram outlines the structure of communication and routing works in the transit system.



4.3 Station Allocation

Stations are allocated passengers by either the user or the spawning task. To determine the next station to go to, the system computes the weight of each station as the sum of the weights of the passengers at that station. Any available trains are sent to the station with the heaviest weights (passengers initially have a weight of 1). When a train is in route to a station, that station is marked as serviced, and any other trains currently in the system will not go to that station until it is no longer being serviced. Sensors that have stations at them are marked as active, while sensors without stations at them are marked as inactive; this is used by the system to determine which sensors to stop at our not. Once no more stations have passengers at them and all trains have unloaded all their passengers, the system waits until new passengers are added before sending a train off. Routing between stations uses our shortest path algorithm.

4.4 Passenger System

Each passenger in the system has their own destination. Destinations are assigned randomly when a passenger is created at a station; a passenger may already be at the station of their choice, in which case they are simply done. Each passenger has an associated weight which begins at 1. When determining which station a train should go to next, the system computes weights for each station as the sum of the weights of the passengers who want to go to that destination; the largest weighted station is the next station that train is routed to. When a passenger is on a train and arrives at a station but does not get off (it is not their station), their weight increases by 1. This allows the system to account for situations where there would be only one passenger wanting to go to a particular station, and as a result, they would never get off because the other stations would be valued higher. In addition, as a part of the passenger system, passengers also communicate to voice their opinions on the service to a channel that can be viewed by running the command "intercom" from within the `Shell`. These get progressively billigerant the longer the passengers are on the train without reaching their destination.

4.5 Viewing State

To tools are provided to the user to view the state of the transit system via the command-line:

Command	Usage	Description
`probe`	`mqchenrtfolks $ probe`	View the stations and trains, and their passengers.
`intercom`	`mqchenrtfolks $ intercom`	View what passengers are saying on the trains.

5 Known Errors

- When trains are lost, they are re-added to the system be first destroying themselves then calling attribution, as a result, the reservations held by this train are lost, which would allow a nearby train to collide with it.
- Trains do not stop at intermediate station when in route to the heaviest weighted station.

6 MD5

```
8afa04fd4ff12bc483271286d52dfa00  /u8/hkpeprah/cs452-microkern/bin/cs452-upload.sh
b79c639122e8ba322f90663c0af9e567  /u8/hkpeprah/cs452-microkern/bin/md5.sh
7d3d938f3360ca46d07b07d6fed3711c  /u8/hkpeprah/cs452-microkern/bin/profiler.sh
40e6f5862869392d9733ea2d6defbb68  /u8/hkpeprah/cs452-microkern/include/bwio.h
bfcb67b08f11ead2eb221f89218918ff  /u8/hkpeprah/cs452-microkern/include/mem.h
41576b24f41f55adf168951b9b5ec580  /u8/hkpeprah/cs452-microkern/include/string.h
03e27f3cd9320300b2b8ffa3c02b9160  /u8/hkpeprah/cs452-microkern/include/syscall.h
daa102a56425b3a9014b297de92855ca  /u8/hkpeprah/cs452-microkern/include/task.h
91425c50507432ecf2bfc92ae70589c4  /u8/hkpeprah/cs452-microkern/include/ts7200.h
6e3a610397f1dc495d4501d15c8efc30  /u8/hkpeprah/cs452-microkern/include/types.h
d898edf77661ac9a98e2a0c6b9d9b9a6  /u8/hkpeprah/cs452-microkern/include/vargs.h
b87cdcf163772da8a5c6f8c9480f407e  /u8/hkpeprah/cs452-microkern/include/k_syscall.h
44fcaefaec7707e3965ce79b0a499df5  /u8/hkpeprah/cs452-microkern/include/kernel.h
73b33f018cb4b0d24d537e14344c9cfa  /u8/hkpeprah/cs452-microkern/include/stdio.h
776cd1e31221f5060f3b30fb4fe8d7e3  /u8/hkpeprah/cs452-microkern/include/stdlib.h
d8ade060b6f468da984678361afbb26b  /u8/hkpeprah/cs452-microkern/include/syscall_types.h
6c827852092f1096308414133a384a4a  /u8/hkpeprah/cs452-microkern/include/term.h
9e8d042c9f18d836b114a1f36de0eb68  /u8/hkpeprah/cs452-microkern/include/utasks.h
628ab30acc9e301a2c45dea456aecd1d  /u8/hkpeprah/cs452-microkern/include/calibration.h
625dbccadf1651555360aa1f36a4da7c  /u8/hkpeprah/cs452-microkern/include/clock.h
66d6b2ea2890fa84bed8c99d92342ecd  /u8/hkpeprah/cs452-microkern/include/train_task.h
58535956378bea20fe03e1974dbfe44c  /u8/hkpeprah/cs452-microkern/include/hash.h
3e840b9eca988d029bdd17167b2693d9  /u8/hkpeprah/cs452-microkern/include/idle.h
7cb6397fc4af9f54ac2bf6ba897ee4bf  /u8/hkpeprah/cs452-microkern/include/interrupt.h
cd31bb05b0e8cdd7f5fc4e216615d9a2  /u8/hkpeprah/cs452-microkern/include/logger.h
8620eebe2b07fb6e0e437c76c9169e0d  /u8/hkpeprah/cs452-microkern/include/null.h
e15aa23da0672fbd1aa9c1ac9c65ac7c  /u8/hkpeprah/cs452-microkern/include/path.h
717b31cadb3b90f022dc0690530d1aab  /u8/hkpeprah/cs452-microkern/include/perf_test.h
f5053c8d3aa93f46ab98e866c16e7876  /u8/hkpeprah/cs452-microkern/include/random.h
7ff8faa9d929453fa8cd82d0e7c32b19  /u8/hkpeprah/cs452-microkern/include/rps.h
840f3b8825b5c29452f00e39d41bc18b  /u8/hkpeprah/cs452-microkern/include/sensor_server.h
21519cac55397ad4c69970d00978d733  /u8/hkpeprah/cs452-microkern/include/server.h
ce8542472ac5ee1d570df56c365f4f12  /u8/hkpeprah/cs452-microkern/include/shell.h
a802e18e57cd94546794499c5fa21226  /u8/hkpeprah/cs452-microkern/include/track_reservation.h
c4025c03395fa5c00f3e688a2c34282a  /u8/hkpeprah/cs452-microkern/include/track_node.h
87ceb5c16ab3bc75696e8f4db59cd20b  /u8/hkpeprah/cs452-microkern/include/train.h
bb3f3f729d453cbd4aeed2947e52f2e6  /u8/hkpeprah/cs452-microkern/include/track_data.h
2bf9ca450d339f34aac9c854c3f8ca7d  /u8/hkpeprah/cs452-microkern/include/train_speed.h
cde78c291020b6e91c68270418d4c5a0  /u8/hkpeprah/cs452-microkern/include/dispatcher.h
89e35e2cf35d247f24787d12aaaa55e3  /u8/hkpeprah/cs452-microkern/include/uart.h
3c918bcb4c46298226d0d896650ad6d7  /u8/hkpeprah/cs452-microkern/include/util.h
af95febea2c3f39c6561dfbd5a4800cc  /u8/hkpeprah/cs452-microkern/include/conductor.h
c54f010d2008b5b5319e2b4ff169a507  /u8/hkpeprah/cs452-microkern/include/traincom.h
d18ed4bc8a21d2c3000476ad5ee393a9  /u8/hkpeprah/cs452-microkern/include/
    train_speed_measurements.h
de27c7043da8ffa3589956bce33e0ffe  /u8/hkpeprah/cs452-microkern/include/demo.h
6a8a106cbe75e45eefe55f44672d7298  /u8/hkpeprah/cs452-microkern/include/sl.h
7098f11916d08562be3e2678d2a0f4ef  /u8/hkpeprah/cs452-microkern/include/transit.h
8402c682ff31b15549689baa00ea45b6  /u8/hkpeprah/cs452-microkern/lib/libbwio.a
aed6cb5a66c843b06e4578283b2a151a  /u8/hkpeprah/cs452-microkern/Makefile
538d526009165606b37f05dafb92ea95  /u8/hkpeprah/cs452-microkern/speeds/data/56.dat
5d76b3044654ce1d8ee5a5398679dd4d  /u8/hkpeprah/cs452-microkern/speeds/data/45.dat
3089bec88fc956944c48d7935380bf75  /u8/hkpeprah/cs452-microkern/speeds/data/47.dat
e16e09dbc121089000fea8485fee5d2d  /u8/hkpeprah/cs452-microkern/speeds/data/48.dat
6aaaed3e895cd9b6b9e940d5714144eb  /u8/hkpeprah/cs452-microkern/speeds/data/49.dat
94ab7bd69e277e89cca00d83aaff8f19  /u8/hkpeprah/cs452-microkern/speeds/data/50.dat
6e4af11d3bb609dc51bbf206b842527d  /u8/hkpeprah/cs452-microkern/speeds/data/51.dat
7fd2b8b364b5a643dddee81bba298f50  /u8/hkpeprah/cs452-microkern/speeds/data/53.dat
e697963fbf4b39a7db7af3b7c291c43a  /u8/hkpeprah/cs452-microkern/speeds/data/54.dat
1290b8a81032cbad4d2a97f60580af8e  /u8/hkpeprah/cs452-microkern/speeds/data/58.dat
3082d977b47de11e9953beea69b5f362  /u8/hkpeprah/cs452-microkern/speeds/data/59.dat
```

```
b2a5b1fd1c703216d50d4bc7f3f30516  /u8/hkpeprah/cs452-microkern/speeds/train.tmpl
3de28be3300e942657e6bc163d7665fc  /u8/hkpeprah/cs452-microkern/speeds/parse_speeds
8da00e714e5f8f92edb2fdce848f750b  /u8/hkpeprah/cs452-microkern/src/mem.c
f0ae274d5e80356c7ed6a5933177136c  /u8/hkpeprah/cs452-microkern/src/orex.ld
9f57296e1076464019629bd9d73c51b5  /u8/hkpeprah/cs452-microkern/src/string.c
e4813b07b97fdd5d901de20bd52a5919  /u8/hkpeprah/cs452-microkern/src/syscall.c
691229861b6bbe33e8424e1a37b1af85  /u8/hkpeprah/cs452-microkern/src/task.c
93cd49979801e27687843502c33a9549  /u8/hkpeprah/cs452-microkern/src/k_syscall.c
2cdb2054bb7a3c3ea8d89037a6ccab7b  /u8/hkpeprah/cs452-microkern/src/kernel.c
0d41b435c56be1e3da0913253c353e0c  /u8/hkpeprah/cs452-microkern/src/main.c
75af15ba6be691bc727060ea15961408  /u8/hkpeprah/cs452-microkern/src/stdio.c
6f956a655ace8fcd0b397cba1700cc91  /u8/hkpeprah/cs452-microkern/src/stdlib.c
1f773d2959e09a42e96e4932886ef797  /u8/hkpeprah/cs452-microkern/src/swi.s
233b118b9d20e4cfb863b7b1c5a7141a  /u8/hkpeprah/cs452-microkern/src/hash.c
e458bcfa62bad28ca7744dede24c2d0d  /u8/hkpeprah/cs452-microkern/src/idle.c
0f2ebe70fd21681e731f29012ce26567  /u8/hkpeprah/cs452-microkern/src/interrupt.c
9b9ab0358247b8b603609a84f792f9e7  /u8/hkpeprah/cs452-microkern/src/logger.c
84fac444c52e04b84e461294c9bf61f2  /u8/hkpeprah/cs452-microkern/src/path.c
e5be2b8b6d95d16a9213773ba0ef2111  /u8/hkpeprah/cs452-microkern/src/track_node.c
1577772814d3c89be20c20352247c7f3  /u8/hkpeprah/cs452-microkern/src/random.c
fc4cef6ae946ff01f4295c992e274070  /u8/hkpeprah/cs452-microkern/src/servers/clock.c
c952a04d585922632ed1e1d1695fb4de  /u8/hkpeprah/cs452-microkern/src/servers/sensor_server.c
273067bd5900fa4955ef29c3c8ee58a2  /u8/hkpeprah/cs452-microkern/src/servers/uart.c
128ac92ea6ed1c5ec6aa34d5d93435a2  /u8/hkpeprah/cs452-microkern/src/servers/server.c
cb7ba0636124c549891a9d5fa7c847f5  /u8/hkpeprah/cs452-microkern/src/track_data.c
ddea7809fe5c18b296d47c7798118da6  /u8/hkpeprah/cs452-microkern/src/tasks/null.c
a3e809bb354610a24e62a6b42804f9e9  /u8/hkpeprah/cs452-microkern/src/tasks/rps.c
5529785e206716ea37759e99b45bae20  /u8/hkpeprah/cs452-microkern/src/tasks/shell.c
7a7baa74de531fa2eff569956f920a14  /u8/hkpeprah/cs452-microkern/src/tasks/utasks.c
06c2629f60253089bde188ae4b4bd7bf  /u8/hkpeprah/cs452-microkern/src/tasks/demo.c
aee0d29b7fa5ff9579fc0962371e8c44  /u8/hkpeprah/cs452-microkern/src/tasks/sl.c
6de7bc338143c6235a19d835c9a9f90a  /u8/hkpeprah/cs452-microkern/src/tasks/traincom.c
196b78e4bb4a9525e0da3cccd9ad110e  /u8/hkpeprah/cs452-microkern/src/tasks/transit.c
6c5117ed2fe8ba9e5ecbc990ab3494a3  /u8/hkpeprah/cs452-microkern/src/train.c
971b3aca602c80e447fdfe74ff177bff  /u8/hkpeprah/cs452-microkern/src/train_speed_measurements.
    c
31a0db625e0980e22a252c28fe7a195a  /u8/hkpeprah/cs452-microkern/src/ui/calibration.c
f8b886a796fdee579142ace0bfe69a56  /u8/hkpeprah/cs452-microkern/src/ui/term.c
5226234b8d0ac893d9aa442e4fe63a5b  /u8/hkpeprah/cs452-microkern/src/util.c
b3300c22dcfded674fed51ccf54ca4ad  /u8/hkpeprah/cs452-microkern/src/train/dispatcher.c
3caf584ca92c177dd3798a309bf92830  /u8/hkpeprah/cs452-microkern/src/train/train_task.c
bc55391fd5f8427c96814da44480e7cd  /u8/hkpeprah/cs452-microkern/src/train/conductor.c
39c5fa8038a9ce86b71341ebee2edd29  /u8/hkpeprah/cs452-microkern/src/train/track_reservation.c
791c987d5fff2c2264dcd25d95aece29  /u8/hkpeprah/cs452-microkern/src/train/train_speed.c
899147a58ac1ffe69003912e2c92e0fd  /u8/hkpeprah/cs452-microkern/track/README
457b119b65fe2a141ef8027dbd216d68  /u8/hkpeprah/cs452-microkern/track/new/parts_tracka
2ddb7d3318b2da2573df4fa481885c62  /u8/hkpeprah/cs452-microkern/track/new/parts_trackb
ca16045b499e0c8333f32fc5d4822919  /u8/hkpeprah/cs452-microkern/track/new/tracka_new
1674c3cb2e1ce6739d8b910b49b260b0  /u8/hkpeprah/cs452-microkern/track/new/trackb_new
d6d8953ae02f593ffb61b441010d617b  /u8/hkpeprah/cs452-microkern/track/parse_track
9b68c8cea9bd88aad8571cae7f35a03f  /u8/hkpeprah/cs452-microkern/track/tracka
f01263617106358e3fea00e84f9b324a  /u8/hkpeprah/cs452-microkern/track/trackb
```