CS 452 Assignment TC1

Victor Scurtu (20678028) and Alex Danila (20516806)

February 24, 2020

Operating Notes

The executable file is found in the student environment under:

```
/u/cs452/tftp/ARM/vscurtu/tc1/kernel
```

The code can be found in the following repository:

```
https://git.uwaterloo.ca/vscurtu/cs452_kernel
```

To build, use the following commands at the root of the repository on a student environment computer:

```
make
```

To run the precompiled kernel, use the following command in redboot:

```
load ARM/vscurtu/tc1/kernel; go
```

After entering the above command, the program will display state information about the kernel and track and begins an initialization procedure to set the track to a known state. All commands sent during the initialization period are ignored and an error message is displayed accordingly. This is because the track is not yet in a known state while initializing. In order to enter a command, begin typing and press ENTER once you've formulated your command.

On top of the commands required for K4, the following commands have been added:

- `it <train_number> <start_sensor> <end_sensor> <speed>` - Route the train from the current sensor (train is place with the pickup on the side the arrow is pointing) to the end sensor and have it be going with the set speed when the end sensor is hit.
- `sl` - Set the turnouts such that the centermost loop of the track is closed.
- `tp <train number> <end_node> <offset>` – Route the train from the current location to the end node with the desired offset as described in the assignment description. sl and it should be called beforehand to get the train running around the center loop. Also ensure the right track is configured using the st command.
- `st <track_id>` - Switch the active track to "A" or "B".

Ensure that the console window is wide enough to display all the information comfortably. If things look messy, try quitting, widening the window, then rerunning.

Table of Contents

This program description assumes that the report for K4 has been read beforehand. It describes only the set of changes from K4, and does not go into detail about kernel components and user tasks implemented in it.

Terminal Output Handling

Terminal Printer

A new task `terminal_printer` has been added in terminal.c that acts as a separate organizer for prints. It's registered in the name server as `terminal_output`. Previously there was an issue with the terminal sending commands to tasks that could then request a terminal print, blocking both tasks. To remedy this, the terminal no longer acts as an organizer for prints, and the functionality has been offloaded onto terminal_printer.

`TPrint` and `TPrintAs` are the functions that all other user tasks should call to print. They format the messages passed to them and forward the data to `terminal_printer`, which is a proper server that does not make any sends, and thus eliminates the possibility of a cycle that was present previously.

`TPrintAs` is new in TC1 and acts the same as `TPrint` with the addition of taking in the coordinates to print the string on the terminal.

Terminal Output

Track Output

A snazzy visualization of the track has been added to the terminal output, which updates when using `sw` commands and visually represents the flow of trains on the track. A print has been provided for both track A and B that updates when switching tracks with the `st` command. Unfortunately, state is not currently maintained across the train controller and so commands done automatically by the train controller don't update the visuals. Thus, the diagram can only currently be used to set initial track conditions with `sw` commands. A common state for switches that derives from the `tc_server` will be added in TC2 to fix this.

When calling the `tp` command, while the train is following its path, the requested next sensor expected, and time/distance deviations from expectation are printed. It's worth noting that this is currently only done while path planning is occuring.

Train Control Server

The primary task added in TC1 is the train_control_server. It is defined in `train_control_server.h` and is registered under `train_control` in the name server. The job of this task is handling high level train command procedures, in the case of TC1 this is mainly path planning and stopping.

On startup, the task precomputes all shortest paths between every set of nodes on the tracks using Dijkstra's algorithm. The results are stored in two dimensional arrays that can be indexed by the source and destination notes. These store both the "previous node" and "distance" values discussed in class. By default the previous arrays are initialized to -1, providing an easy way to check if a path exists between two nodes.

The server provides two main commands, `INIT_TRAIN` and `TARGET_POSITION` which are used to initialize a train to some location and velocity from a starting position, and stop at a target location after being initialized to run in the center loop respectively.

The trains state is stored in a `TrainState` variable. The bulk of path planning work done is by the `TrackPathPlan` structure and its helper `update_path_plan`. `TrackPathPlan` contains a pointer to a `TrackState` and tracks its progress along a path stored inside.

Most pathing work is done after receiving a sensor command, in which case if a new sensor was hit the `TrainState` and the `TrackPathPlan` are updated, and the `update_path_plan` function also switches any necessary tracks. The tracks to switch are decided by looking at all branches in between the next two sensors ahead of the train. This was done in this way so that trains have a minimal impact on the rest of the track, hopefully making it easier to route multiple trains in TC2.

When picking a target position, if a path is immediately available but there is not enough room to stop if the train were to immediately switch off the tracks, it will loop again and begin stopping from a distance further back. Internally, it is creating a new route. Additionally, if anything happens such that the train goes off route, it will attempt to reroute if a new route is available.

Command events can be created that wait a set amount of time, then send a message to the train control server triggering an action. Stopping is done by calculating the exact position where a stop command should be sent at the last possible sensor hit that still has room to stop. An event that triggers a train stop is then created with an appropriate delay. When it's delay expires, it will

send the command to stop the train, possibly somewhere between two sensors such that it stops at the correct location.

Miscellaneous Changes

- The maximum number of priorities has been increased to 16 to allow for the additional notifiers and train control server.
- `MEMORY_START` has been push to a larger address to allow for more global storage required by the precomputed dijkstra's paths.

Known Bugs

- We'll see in the demo :)
- Listen, it's 9:47 and we haven't slept. We can hardly recall the bugs, let alone describe them and all their various causes.

Calibration Testing Methodology

Timing was done for each train on each track. The trains were run for several circuits of the track at three speeds each: `8`, `11`, and `14`. The raw data collected is the time when the sensor dump was requested that resulted in a new sensor being tripped, in milliseconds of Unix Epoch time.

In addition, two custom built laser trip wires were also made and included in the logging data as `L0` and `L1`. The laser trip wires were used due to their low latency (virtually 0) and their freedom to be arranged anywhere on the track. Specifically, they were arranged a known distance apart (2 cm) so that the instantaneous velocity of the train could be estimated. The laser sensors were driven by an Arduino. The raw data for the lasers is in milliseconds since the Arduino powered on.

In order to reduce the dependence on polling a laser sensor at the correct time, the Arduino was programmed to record the time at which each sensor was tripped and respond with that time when queried over USB serial. This ensures virtually no latency in the time readings from the laser sensors.

In order to consolidate the disparate sensor types, the Marklin control was connected directly to a personal laptop that was also connected to the Arduino controlling the laser sensors. Various Python scripts controlled the track and polled for sensor data from the Marklin and the Arduino.

Global average speeds for each track were taken by running the train around a circuit of known length - 4867mm for Track A, 4698mm for Track B. The train was allowed to accelerate, then

the time it took for the train to come back around to each sensor in two cycles was measured. Each test was run several times, but only recorded twice. Due to the low latency of the laser sensors, the times observed were fairly consistent.

Stopping tests were done at the conclusion of loop tests. Stops were conducted in a straight line, and two distances were recorded and averaged for the kernel. Stops were triggered by the laser sensors, again due to their low latency. The instantaneous speed as measured by the lasers was also recorded for each train and combined with the distance of the stop to get a rough deceleration constant for each speed. This is not currently used but may be used for TC2.

Raw and analyzed data, and code are in the `data` and `code` directories of the `calibration` folder respectively.