CS 452 Kernel 3

Brian Forbes & Daniel McIntosh
20617899 & 20632796

June 27, 2018

1 Building, Running, and Usage

The repository for the code can be found at `https://git.uwaterloo.ca/baforbes/cs452-kernel/tree/tc1`. To build the executable, run the following command from the root directory: `make` To build this document, run the following command from the root directory: `make docs` The executable will be built to `<repo_root>/bin/kernel.elf`

Once run, the program will initialize the kernel, and then create the first user task. This task will then create the tasks that run our train program, such as the name and clock servers, UART handlers, and the terminal, command, and track state servers.

2 Program Description

Note that each referenced file corresponds to a header file of the same name in the `include` folder (for example, `src/kernel.c` to `include/kernel.h`)

2.1 Track State Server

The track state server is in `track_state.c`. To keep track of the state of the track, we added a Track State server. This keeps track of triggered sensors, switch states, train states, and the map of the track. On initialization, it loads the track map from `track_data.c`, based on which track it is.

2.1.1 Stored Track State

The sensors, switches, and track map are stored in arrays. When a command is sent to switch a switch, it is updated in the Track State server. As well, when the sensors change, they notify the Track State server.

The Track State server also stores values related to the state of the current train. This includes the current speed, the current predicted velocity, the last triggered sensor, the time of the last triggered sensor, the distance between the

last triggered sensor and the next expected triggering sensor, the expected next triggering sensor, the time expected to hit the next expected triggering sensor, and the error (in mm) between the last triggered sensor and the expected time of the last triggered sensor.

2.1.2 Routing

Route requests are sent to the server in the form of an object (sensor, switch, merge, or exit) and a distance past that object, where the result is a list of switches that must be changed, a sensor to wait for, and a delay after that sensor to stop the train. To find a route, first a DFS is used from the next expected sensor to the target sensor. Then, if the distance past that sensor is greater than the stopping distance, an additional forward search is used to find the last sensor before the stopping distance. If the distance is less, it searches backwards through the route to find the first sensor greater than the stopping distance to wake up on. Using this sensor, it calculates (using the current predicted velocity) the expected time between the sensor triggering, and the need to stop the train. Then, it returns the path discovered by the DFS (in the form of a set of switch values), the sensor it found through the above process, and the calcualted time.

2.1.3 Velocity Calibration

We perform velocity calibration on an ongoing basis. Every time a sensor is hit, the velocity is updated (in the form of a moving exponential average with $\alpha = 15$) with the velocity calculated between the last two sensors. Velocity is stored in a table, corresponding to each speed. This allows the velocity prediction to stay relatively calibrated when speed changes, instead of taking time to re-calibrate on each speed change. In addition, when sensors are hit, the data related to the train discussed previously are all updated. Based on the current track state, the next sensor is predicted, and the distance to that sensor is calcualted. Then, based on the updated velocity, the time to reach that sensor is predicted.

Any time that a sensor triggered is not the calcualted next sensor, that measurement is dropped and the calibration begins again once the next, correct sensor is hit. This allows the program to handle sensors that are not working, or cases where the stored track state does not match reality.

2.1.4 Stopping Distance Calibration

2.1.5 Short Moves

To facilitate short moves, a table of speeds and delays was manually calibrated. This table is in $20mm$ increments, between 1 and 219 mm. When a short move request is sent to the track state server (containing a distance), the server returns the speed to start the train at and the time to wait before the train is stopped. This allows the train to move in short increments.

2.2 Train Event Server

2.3 New Commands

In addition to the commands previously implemented in K4, several new commands were implemented to allow the user to route the train to various track elements.

2.3.1 route

Route takes three arguments (a track object, a distance past that track object, and a train) and routes the train to a point that distance past that track object. It does this by retrieving a route from the Track State Server, which is returned in the form of a list of switches to change, a sensor, and a delay time. It then switches all required switches, and begins a task which is notified once the required sensor is triggered. After this notification, it waits the given delay, and then sends the stop command. This delay is calculated based on the velocity and stopping distance of the train. There are 4 types of objects that can be routed to:

1. Sensors Sensors can be routed to as $[A-E][1-16]$, corresponding to the labels on each sensor.
2. Switches Switches are routed to as $S[1-18, 153-156]$, corresponding to the labels on each switch.
3. Merges To route to a merge (a switch, but in the opposite direction), $M[1-18, 153-156]$ is used.
4. Exits To route to an exit, use $X[1-10]$.

2.3.2 cal

2.3.3 move

Move takes two arguments: A distance (between 0 and 219, in millimeters) and a train to move, and moves that train forward approximately the given distance. This is by requesting a short move from the Track State server (which returns a speed and a delay time), starting the train at that speed, and then starting a worker which delays the given time and stops that train.

2.3.4 param

Param takes three arguments: a distance (whose parameter is to be changed), a parameter (either 'D' for Delay or 'S' for Speed) and a new value, and updates this parameter in the short move lookup table for the given distance. It is used for calibration, and should not be used during normal operation.

2.3.5 inv

Inv switches every switch to the opposite of the current state. It is mainly for debugging as well, and should probably not be used during normal operation, but it is left in in case a user wishes to invert all switches.

2.4 New Data Structures

The majority of all data structures added were arrays or circular buffers, used in many places. We also added a large number of different message and data storage structures, none of which are particularily complicated.

2.5 Changes to the Kernel

One new syscall was added to the kernel: CreateWith2Args, which is used to create a task with two initial arguments, allowing the task creator to save the send of data to the created task. It is used to implement various short-lived workers. It works by setting the `r0` and `r1` registers when initializing the task.

As well, the kernel now disables interrupts in the VIC when exiting. This fixes an issue we encoutered preventing our program (and other programs) from running after we quit without resetting the arm box.

3 Explanation of Program Output

3.1 Switches

3.2 Sensors

3.3 Command Terminal

3.4 Status Bar

3.5 Data Display

At the top of the display, idle time, last error in (ms), and last error in (mm) are displayed. At the bottom of the display, various debugging data is printed.

1. `STK_LIM`
2. `STK_MAX`
3. `STK_AVG`
4. `VELO_PR` The predicted velocity of the train, in terms of $\frac{cm}{10000s}$, which is just $\frac{cm}{s}$ multiplied by 10000.
5. `SNSR_NX` The next sensor the program expects the train to hit.

6. `DIST_NX` The distance between the last sensor hit and the next expected sensor.

4 Known Limitations

1. We do not calculate the shortest path between nodes.
2. The route command requires the train to already be moving at speed to travel to a node
3. We can only calculate linear routes - the route command does not understand the concept of reversing.
4. No care is taken to ensure that the switches are not switched when the train is on top of them, resulting in derailment if care is not taken with the timing of a route command.