CS 452 Kernel 3

Brian Forbes & Daniel McIntosh
20617899 & 20632796

June 28, 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 Clock additions

There is now a Timeout function which will sawn another task to send a TIMEOUT message to the calling task after a specified delay.

2.2 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.

2.2.1 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.

2.2.2 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.

2.2.3 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.

2.2.4 Stopping Distance Calibration

We start with an approximate stop distance calibration table for the various speeds, which is then updated when the user runs the `cal` command. The calibration sequence is as follows:

1. In a loop of size 8:
   (a) Get a route to the sensor from the Track State Server, a route is described in greater detail under the `find` command.
   (b) Set the necessary switches, just as in the `find` command
   (c) Wait for the sensor specified in the route to be triggered
   (d) Wait the delay specified in the route
   (e) Send a stop command
   (f) Wait for the destination sensor (given as a parameter to `cal`) to be triggered, or a timeout of 2.5 seconds
   (g) Notify the Track State Server of whether or not the sensor was triggered, and what iteration of the loop we're on, so it can adjust its stopping distance
2. if every iteration of the previous loop ended on the same side of the sensor (i.e. the sensor was always triggered, or never triggered), restart.

Since the Track State Server uses the stopping distance in it's calculations, every iteration of the loop will result in a route with a different delay, and potentially different sensor. The stopping distance is adjusted by a base value of 70, decreasing by a factor of 4/5 for each iteration of the inner loop, resseting if we re-start.

2.2.5 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.

2.3 Train Event Server

The train event server provides `TrainEvent_Notify` and RunWhen calls. `TrainEvent_Notify` is a wrapper around a send to the train event server, and is called by Track State to alert the train event server of sensor events. RunWhen allows other tasks to preform predefined actions when a sensor is triggered. It takes a sensor, a pointer Runnable struct and a priority, and spawns a new task with the specified priority, with the sensor as an argument, then sends the Runnable struct to it. The Runnable struct contains a function pointer, arguments, a timeout

and `run_on_timeout` boolean. The spawned task will call the function with the given arguments if the sensor is triggered before the timeout period. If the timeout period expires, and `run_on_timeout` is true, the function is also called. To help distinguish between a timeout and sensor event, in addition to the given arguments, the function is passed a boolean to indicate whether it was called because of a timeout or sensor event. This is all accomplished by a call to the clock Timeout function, and a send to the train event server. The train event server maintains an array of tid's waiting for a sensor. When the train event server is notified of a sensor event, it then checks if a task is waiting on that sensor, and if so sends a WAKEUP message to it. The task RunWhen spawns queues in the train event server, and in the case of a timeout unqueues.

2.4 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.

2.4.1 find

Find 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 (using RunWhen from the Train Event Server). 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 $X[1-10]$.

2.4.2 cal

Cal takes two arguments: A sensor and a train, and initiates the calibration sequence on that sensor.

2.4.3 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.

2.4.4 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.

2.4.5 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.

2.5 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.

2.6 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

There is a list of the current state of all the switches on the left side of the terminal

3.2 Sensors

Any time a sensor is triggered, we print it to right side of the terminal. Eventually we loop back and start printing them at the top again.

3.3 Command Terminal

There is a prompt in the middle of the terminal, where the user can input commands.

3.4 Status Bar

At the bottom of the terminal (just below the switch list), we output a small status bar. Currently it has 4 flags, 3 of which are implemented: `(I)nvalid command`, `(F)inding location on track`, and `(C)alibration running`.

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` The maximum possible stack size for a task (i.e. the size at which we start to overwrite another task's stack) - this is only calculated (and printed) once at startup, since it does not change
2. `STK_MAX` An ongoing maximum of the stack sizes of task - a task's stack size is only checked during a syscall, so this is only approximate
3. `STK_AVG` A rolling average of stack sizes - similar to STK_MAX this is only approximate, but still accurate enough to alert us if there is a problem
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.