CS 452 Project

Brian Forbes & Daniel McIntosh
20617899 & 20632796

July 29, 2018

1 Building, Running, and Usage

The repository for the code can be found at `https://git.uwaterloo.ca/baforbes/cs452-kernel/`. 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, track state, and train state servers.

2 Introduction

Our goal in the project section of the course was to implement a more successful TC2 than we had for the TC2 deadline. In the end, we had a complete reservation system, the ability to find, execute, and recalculate routes, and error handling for missed sensors and incorrect switches.

3 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`)

3.1 Track State Server

The track state server is in `track_state.c`. It handles the current state of the track (switches and sensors), and handles route finding. When it receives the result of a sensor query, the sensor server (`src/sensor.c`) sends a notification to the track state server. In turn, when a sensor goes from 0 - 1 (i.e. is newly triggered), the track state server notifies the Train server of the sensor, so it can attribute it to a train.

3.1.1 Route Finding

Routes are now expressed as a sequence of switches and associated actions. Actions can be either switching to curved, switching to straight, or reversing. In the case of a reverse action, the switch is inferred to be a merge, instead of a switch.

When a route is requested from the track state server, it is given the start and end nodes, a bitfield representing unusable nodes (due to reservations), the minimum distance of the route, a maximum distance to search, and the distance penalty for reversing. Then, a variant of BFS is performed where possible paths are stored in a minheap. In this way, the first path found will necessarily be the shortest path. The impementation of the BFS is in `track.c`, as `find_path_between_nodes`.

In our project, we decided to apply a generous penalty to reversing, because we considered reversing to be both slower and more error-prone compared to forward routes. As such, trains tended to only reverse when their way was blocked, or it was significantly shorter to reach a destination by reversing.

3.2 Train State Server

The train state server is in `train_state.c`. The train state server now handles most of train control, including sensor attribution, reservations, and the execution of routes generated by the track state server.

3.2.1 Sensor Attribution

To attribute sensors to trains, when a sensor is triggered, the train state server attributes it to the train which has the shortest path to that sensor. This includes a lower bound of distance 1 and an upper bound of 3000. The lower bound prevents us from re-attributing successive sensor hits to the same train when 2 trains are running close together, and the upper bound prevents spurious sensors from being attributed to a train. Also note that despite using the same function as route handling, the shortest path calculation for sensor attribution does NOT consider reservations, as this would cause problems if a switch failed in such a way that a train ended up on track reserved by someone else.

3.2.2 Route Handling

The train state server stores Active Routes for each train. These contain the route found by the Track State server, and various pieces of data about the train's current place in that route. As a train follows its route, it reserves the track ahead of it. Each time a sensor is triggered and attributed to a train, any steps of that route which are on track newly reserved by the train are executed. This way, switches are not changed until the train has reserved the track they are on. This also includes stopping. Once a train has reserved the piece of track it intends to stop on, it calculates the expected delay until it needs to stop. Because our reservations were very far ahead from the train, it often

has completed its entire reservation by the time it hits the first sensor. As a result, we programmed an acceleration model to approximate the trains velocity, thereby making the train more accurate, and preventing us from sending the stop command immediately at the start of a route.

Routes are stored as a list of actions. As such, the train state server stores two indexes in this list for each route: the index of the next part of the route the train will pass ("current position index"), and the index of the command latest in the currently reserved section of track ("reserved index"). Each time a sensor is hit, the current position index is updated to match the location of this sensor. The current position index is used to project the position estimation forward in the route, in addition to calculating distances to the end of the route, the next action in the route, and the distance to the next required stop in the route. The reserved index is updated when actions are performed. It is used to determine the actions to be performed when track is reserved.

3.2.3 Position Estimation

The position subsystem is in `src/position.c`. To model the position of the train at any given time, we store the state of the train (constant velocity, accelerating, decelerating, or stopped), the last known node of the train and the time at which it was at that node, the current velocity, acceleration, and maximum velocity, in addition to, when the train is stopping, the expected end of that stop.

When the state of the train changes, the train server updates the state in its position struct. When the position needs to be calculated, the delta time between the last known position and the current time, dt, is calculated. Then, the portion of dt where the velocity was changing is calculated using the equation $v_f = v_i + at$ (where $v_f = v_{max}$ if accelerating, and $v_f = 0$ if decelerating). Then, the distance travelled while accelerating is calculated via the formula $d = d_0 + v * dt + \frac{a*dt^2}{2}$. Then, the section of the time at constant velocity is calculated by subtracting the acceleration time from dt, and the distance travelled at constant velocity is calculated via $d = d_0 + v * t$. This total distance is projected forward from the last known node to determine the estimated current position of the train.

Similarly, to determine stopping delay, the current velocity can be queried from the position struct (which is calculated as $\mathbf{MIN}(v_{max}, v + a * t)$.

3.2.4 Reversing

To reverse a train, similar actions are taken to stopping. After the stop has completed, the train state server is notified. It sends the command to reverse the train, and then dispatches a task to re-accelerate the train after a very short delay.

3.2.5 Short Moves

When the distance from a stop to the next stop along a route is less than 1000, we use the short move subsystem. It operates similarly to the reversing action, where after accelerating the train to speed 14, a worker is dispatched to notify the train server it stop the train after a delay. To calculate the short move delay, we first built a quadratic model to represent the distance travelled by the train based on the delay after which we stopped it. We then inverted the function to get a function from distance required to delay. We chose a quadratic function for this, as it was represents a constant acceleration model, which is reasonably accurate to the trains. As well, a quadratic function has a much simpler inverse compared to a cubic or higher degree function.

3.2.6 Acceleration

Because we did not calculate acceleration manually, we instead calculated acceleration based on the calibrated stopping distances. Using the equation $v_f^2 = v_i^2 + a * d$, given that we knew v_i, v_f (0, since the train is stopping), and d (the stopping distance), we could calculate a. This model assumes that the acceleration is constant, which we do not believe to be the actual case, but it is still reasonably accurate. It also assumes that the acceleration and deceleration are the same, which we determined emperically to be false. To rectify this, we multipled the acceleration by $\frac{3}{10}$, which we determined empericaclly to be a reasonble approximation of the actual difference between acceleration and deceleration over medium distances.

3.2.7 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 $\alpha = 0.15$) 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.

3.2.8 Reservations

Once a navigate command has been given, a train begins to reserve track in front of it. Reservations occur on a node by node basis instead of edge by edge. A train reserves track nodes 2 sensors + stopping distance + next switch ahead of itself. This guarantees that we have enough space to stop if we are unable to reserve more track, and a sensor doesn't fire as expected. It became apparent in testing that this was excessive and unnecessary. Ideally we would have changed to reserve only (timeout * (vel + error margin) + stopping distance), where our timeout is the expected time it takes us to hit the next sensor + an error margin. A train would have tried to reserve more track when either the timeout or the next sensor fired, and stopped if it couldn't reserve more. Since the timer is

guaranteed to fire, a faulty sensor doesn't risk causing a collision. Unfortunately we ran out of time to do this.

We free track behind us as we hit sensors, excluding the sensor we just hit. This is enough to guarantee collision avoidance, because reservations are inclusive of their emergency stop points (2 sensors + stop dist or 1 sensor + timeout margin + stop dist).

Reservations are stored as an array of 6 bitfields of size 128 (split into 2 long long's), and a cumulative/total bitfield. This is enough for 1 for each potential active train on the track we can handle (5), and 1 for user reservations. Convinently, this is the same as the number of colours available on the terminal. Also note that we only needed 124 total bits because we don't consider enter/exit points for reservation, because the assumption is that you own the node before, and there is not enough space on the track for 2 trains between an enter/exit and the adjacent node.

Reservations for an individual train are passed around within the `train_state` server a pair of pointers, 1 to it's reservation bitfield, and 1 to the total bitfield. This improves separation of concerns, since methods are not given access to information about other trains. The track nodes a train is forbidden from are calculated by XOR'ing the train's bitfield with the total bitfield. This result is used before reserving track, and passed to the `track_state` server when asking it to generate a route for a train. This also left the `track_state` server largely unchanged from what it was at TC2.

3.2.9 Switch Error Handling

When a sensor is attributed to a train that is not on its active route, the error handling subsystem activates. In a single atomic action, it releases all its current reservations, and recalculates a new route to its goal. We chose not to have the train stop immediately when it leaves its route, because we thought that it would be preferable for it to stop only if it was unable to acquire a new route. Because the release and re-route is atomic, the train will, with complete certainty, be able to re-acquire the reservation for track it already owned, nullifying any potential benefit to holding onto the old reservations.

3.3 Data Structures

3.3.1 Minheap

We used the same minheap we previously implemented for our kernel in our routing system, to perform our route serach. The minheap itself stored pointers into a pool of route data structures, with a priority equal to the distance of that route.

The minheap itself was implemented on top of a constant size array. It has three operations: `add`, `remove_min`, `peek_min`.

add The element is added to the end of the array, and then bubbled upwards. This is logarithmic in the size of the array, which is a constant - so this

takes constant time.

peek_min The element at the 0^{th} index is returned. This takes constant time.

remove_min The element at the 0^{th} index is swapped with the last element, and then removed. The new 0^{th} element is bubbled down. The removed element is then returned. Similarly to `add`, this is logatrithmic in the size of the array, which is still constant.

3.3.2 The Track

The first user task now initializes the track into a section of shared memory. There are no concerns about sharing this memory, as it is treated as read-only by all servers, and is therefore safe to be shared. To determine which track is currently in use, the mac address of the device is queried. For track A, this ends in `0xC5`, whereas for track B, this ends in `0xCC`. The other preceding characters are the same for both tracks. This method of determining current track was shared with the class by `c7zou`.

3.3.3 Circular Buffers

In any section of our program where data is buffered, we use circular buffers. This allows us to have aeffectively infinitely-large buffers (as long as reading and writing happen at similar frequencies). Circular buffer were implemented as arrays, with a read and write index. There is also an empty flag, to differentiate between the full and empty cases.

4 Known Limitations

1. The program is not very robust, and crashes very often.
2. The acceleration model is not very accurate (nor are other calibrations), so the train does not stop very consistentlly.
3. If the train overshoots a stopping position, it can sometimes misinterpret sensors to mean it has gone off its route, and will re-route itself back to its destination multiple times until it stops successfully.