CS 450 Assignment K2

1. How To Operate Program

The .elf artifact is committed in the git repository. See section 4 for a link to the code repository. Alternatively see section 5 for instructions to build from source. Once the artifact is obtained, load it into redboot and run `go`.

2. Group Member Names

Xuanji Li

Lennox Fei

3. Kernel Structure

Synthetic Interrupt

The modem status (MS) interrupt is not directly connected to the VIC. This makes me sad because it must be accessed using the combined interrupt, which is also triggered on the TX interrupt. I decided to make the kernel abstract this away by allowing the user program to listen to a "synthetic" interrupt, i.e. an interrupt that is not connected to the VIC. MS is a synthetic interrupt. When a combined interrupt comes in, the kernel checks if the combined interrupt contains MS, and if anyone is waiting for the synthetic MS interrupt, and if so, fires it. User programs are not allowed to wait for the combined interrupt directly.

UART Servers

The UART servers are located in `com1.c` and `com2.c`. I will explain COM1 first. There are 3 notifiers, 1 for each interrupt (RX, TX, MS). The MS interrupt is meant to catch the CTS modem signal. The FIFO is disabled. The TX notifier gets awoken if there is no data in the 1-bit output buffer, and then sends to the COM1 TX server. The server replies to this send when it wants to write data to COM1, and replies with the character to write. If there are no characters to be written to COM1, the TX notifier is send-blocked. The RX notifier gets awoken when there is data in the 1-bit input buffer. It sends the data in the buffer to the RX server. When there is no incoming data, the RX notifier is await-blocked. The MS notifier awaits the MS interrupt and sends a message to the TX server when that happens.

The TX server receives from the TX notifier, the MS notifier, user tasks via Putc, and a special user task called sensorRequestor. It uses the TX and MS inputs to feed a small state machine (after sending to TX, wait for MS to fire twice, and TX to go high, before sending the next character). The user tasks via Putc feed a queue of characters to send. The COM1 server examines every character it sends to the

TX notifier and if it sees any `0x85` it blocks any more sends until the sensorRequestor notifies it that 10 bytes have arrived back. This implements half duplex mode.

The COM1 RX server mantains a queue of things that the RX notifier sends it, and replies to user program that send to it via getc.

Putc can be used in both nonblocking (just insert into the queue) and blocking (wait until the character is sent to the TX notifier), but due to some strange bugs, only nonblocking mode is used.

The COM2 system works exactly the same but MS and CTS are not used.

UI

UI code lives in `ui.c`, but there is some non-UI code mixed in. The UI is provided via the `cli` function, and optional status functions. An example of a status function is clockDisplay; if it is started, it prints the uptime to line 2, and if you don't start it, everything is fine, the UI just does not contain the uptime. The rule is that the optional status functions do not print carriage return newlines to COM2; only CLI is allowed to do that. This way we get a scrolling area as well as an independently-updating HUD.

Since the terminal save/unsave cursor codes cannot interleave, functions that write to COM2 are expected to acquire a mutex before doing so. The mutex is provided via `uiLockServer.c`.

Tracking + Pathfinding Data Structures

There is a function `initPathfinding` provided in `graph.c` that should be called once to initialize global constants. The first one is the track data from the course site. Next it runs Floyd-Warshall on an unweighted graph corresponding to the track data to get all-pairs shortest paths. Once this is done, the lowest-cost route can be read off in $O(N)$ time where N is the length of the route. However this design means that we cannot recompute routes, so this will fail for TC2.

Tracking the Train

`switchServer` tracks and draws the state of the switches, and `trainServer2` tracks and draws the state of the train (speed setting from 1-14, location+direction, sensor timing discrepency vs predicted). You can change the state of a switch from anywhere (manual switching, routing) and trainServer2 will automatically find out about it the next time it needs to calculate the state of the train. Unfortunately this is done by shared memory, but since it is a one way information flow it should be convertible to message passing very easily. We need the switch state to predict the train's future location, but not for pathfinding.

Stop at a landmark

To stop at a landmark, we use the `haltat` command. This queries the trainServer for the current train location, goes to 1 sensor in the future, computes a route to the landmark as well as the distance of the route, and sets the switches position. We pretend that the switches can be set instantly. At the same time it creates a task that waits for a certain amount of time to send the speed-0 command (the amount of time is estimated on the current speed and on experimental data).

Parameter Estimation

Collected data is at http://bit.ly/2JY9PMK

Inter-sensor time is bimodel with two very narrow peaks 100ms apart. 100ms is our polling time (this is printed on our UI). Let us call the phase the distance from the start of a polling window to the time the sensor is triggered. When averaging speed data, the phase averages out to half the polling window, so we can get very precise data automatically. Unfortunately we cannot do the same for stop distance. Estimating the phase of a running train (i.e., to know its position to less than 100ms precision) must be statistical, but I think it is possible. I plan to try implementing phase estimation before doing automatic stopping data measurement, instead of trying to find some way to average out the phase for stopping data.

Error Handling

There are functions `ASSERT` and `PANIC` for unexpected situations. In addition, a lot of things previously done by error codes are replaced by, because it's hard to remember to check error codes in C, and I tried to design my API so that a working train program does not need to check error codes.

4. Code repository

https://git.uwaterloo.ca/x538li/chos2

5. Build and Execution

When you are in the root directory of the repo (chos/), run the following commands

make

make install

Make should build everything, and make install will move it to `ARM/[uwid]`, this allows for parallel testing. In case you get issues with missing linker libraries, do the following to regenerate the symbol link

cd lib ./rebuildLib.sh