CS452: Real-Time Programming

Train Control (Part 1): Controlling One Locomotive

March 8, 2020

Yuhui Pan
20868342

Leanora McVittie
20513649

Usage

The git repository can be found at https://git.uwaterloo.ca/lmcvitti/trains_kernel, and the exact commit hash will be emailed to Professor Karsten; it can also be found as the final commit on the `k4` branch in that repository.

The executable file, `tc1.elf`, can be compiled by cloning the above repository and then running `make` in the root directory. It can be run by loading it into RedBoot and run using RedBoot's `go` command.

A pre-existing executable, `tc1.elf` can be found in `/u/cs452/tftp/ARM/lmcvitti/` and can also be loaded into RedBoot and run.

The executable will load a user interface that displays the switch state, the most recent sensors, the track map, the idle time percentage, the current time, and a console area for typing commands.

Command Line:

`train <train_number>`

Set the train number to begin with.

`track <track_label> ('a' or 'b')`

Set the track to begin with.

`loop`

Make the track to form a loop.

`level <level_number> (0, 1, 2)`

There are three levels of the train, which are 0 for speed 8, 1 for 11, 2 for 14 and their corresponding velocities and distances. `Level` will drive the train to move at the speed of the level.

`goto <track_element> <offset(mm)>`

Route the train to its corresponding destination and offset.

Track Control

Routing

To route the train we use the standard Dijkstra's algorithm. The track elements are stored in a array of structs, and each one knows its neighbors, therefore it is easy to traverse the track graph using this array and to apply Dijkstra's algorithm to the system.

Much more interesting than our implementation of Dijkstra's algorithm is how we handle a number of corner cases that come up when dealing with routing when considering that 1) the start point is in motion while the algorithm is running, and may pass a switch before it can be turned, and 2) for close destinations, it may not be possible to stop the train without overshooting, and therefore an extra go-around is necessary.

The solution in both cases is fairly similar: we wait until the train has passed the problematic area and then rerun the algorithm to reroute the train. There is a safe distance where the program would never switch the turnouts and that is calculated based on the current velocity of the train. The safe distance for higher speed trains will be longer than that of lower speed trains. If it can not switch turnouts when the program figures out the shortest path, then wait till the train pass the next sensor and rerun the algorithm, and this is a repeated process to make it absolutely safe to switch the turnouts, and we have spent time in calibrating the right safe distance in terms of train's current velocity. When rerouting, the program is in 'reroute' mode and there is a lock to make turnouts switch safe, which will be unlocked when the train has passed the next sensor.

The switch commands are atomic which means either the routing is successful and switch all the turnouts, or no turnouts will be switched and we rerun the routing algorithm. Besides, the velocity is dynamically calibrated and would be used with stopping distance to get the delayed ticks for issuing the stop command.

Stopping

The timing to stop is calculated based on the formula:

```
delayed_ticks = (distance_to_destination + offset - stopping_distance)/current_calibrated_velocity
```

When `goto` command is issued, the program will enter a 'goto' mode if it finds the route and all turnouts can be safely switched. The distance to destination for each sensor that will be visited through the shortest path will be pre-populated and used to calculate the `delayed_ticks` mentioned above if condition met. For each sensor update, we will check the distance to destination and issue the stop commands if destination can be reached by delaying a few ticks and set speed to 0.

Measurements

Our data is in `data/train_measurement.json`

Velocity

We created a tight polling loop to determine the time it took to go around a full loop of the train set - we chose the inner oval - and then recorded how long it took to get from our chosen sensor around back to that sensor again. We then added up the total distance of this loop, averaged the recorded times around the loop, and found the average velocity. We also automated this calculation through the program so that whenever train speeds were changed, the time average and the final velocity calculation were printed and the collected timings were reset, providing a blank slate for the next speed calculation.

Acceleration

Unfortunately stopping distance measurements could not be automated like the velocity ones could, and because they are so tedious to procure, we got less of them. Because the manual workload is so much heavier, the code used to measure the stopping distances could be significantly simpler, simply issuing a stop command once a sensor was triggered, at which point the distance the train traveled from that sensor can be measured.

Unlike velocity measurements, we chose to execute these measurements from within our kernel instead of within a tight polling loop. The goal here was to simulate the same delays that will occur during actual execution of the train control. While the velocity of the train is unaffected by any processing delays within our executable, the stopping distance would be. Therefore we chose to build this into our measurements by using the kernel and train control code to trigger the stop command.

Variable Idle Time

The only way we could think of to have an exact sliding window for the idle time percentage was to store the lengths and times of all the idle task executions that occur during that window, in order to know when a task execution falls out of the window. Even if we determine an average execution time and track how many idle task executions have occured, all the execution times still have to be stored. A quick experiment showed that we have over `125` separate idle time executions per second. The space that would be required to store all those times would be completely space-prohibitive.

Instead of calculating an exact idle time percentage with a sliding window, we then decided to approximate it with an approach that combines time slicing of smaller increments and the rolling average formula that was discussed in class. We take half-second time slices and calculate the percent idle time within that time slice. After that time slice is concluded, the idle time calculated within that time slice is rolled into the average of all the idle time percentages, which is what is displayed as the idle time percentage on screen.

For this calculation, we use an alpha of `(1-1/(2*IDLE_LENGTH)` where `IDLE_LENGTH` is the length of the window of time (in seconds) we're interested in simulating to calculate the idle time percentage. We're using time slices of half a second, so the most recent time slice has the exact same weight as if it were within an exact sliding window. As time slices get farther back, this becomes less true. We decided that this was a reasonable solution for not having the space to store all the data necessary to do it exactly.

Previously Implemented

Input/Output Management

There are eight total IO tasks; a server and a notifier for each of input and output for each UART device. Each task can then be identified with three binary variables: `UART1` or `UART2`, `input` or `output`, and `server` or `notifier`. An example task would be: `uart1_rx_notifier`.

Notifiers

Six new hardware interrupts were created, one for each of the notifiers and two more for `CTS_AST` and `CTS_NEG`.

For the three simple notifiers (both `UART2` notifiers and the `UART1` receiver) the system is simple: the notifier calls `AwaitEvent` and blocks on its event queue, setting the necessary flags in the process. When the event is triggered, the notifier is put back on the ready queue and when the notifier is scheduled again it sends a message to its associated server, either to pass it the received character or to request a character to send.

The `UART1` transmitter is more complicated because of the necessity of checking that the `CTS` flag has both gone down and come back up again before asserting that data can be sent. To ensure that this is the case, this notifier has extra checks. The first check ensures that the `CTS` flag falls before looping back to wait on the `TX` flag. The second check, after the `TX` event has been triggered, ensures that the `CTS` flag is back up before sending a notification to the server. This ensures that the transmission channel is available when the server is alerted.

Servers

Each of the servers behaves in essentially the same way, with the transmitting and receiving servers mirroring each other. The server receives characters (from either its notifier or other tasks) and buffers them to send (to other tasks or its notifier). To differentiate between cases, each message sent to the server includes a `source` field which identifies whether the sender was a task or the notifier. The server then knows whether to buffer the sent data or to reply with a buffered character. If a task requests a `char` when none are available, it holds onto that task id and immediately replies with a `char` when it receives one.

The buffered characters are stored in a circular buffer which is implemented as an array and increments a write index when adding a character to the buffer, and a read index when removing a character from the buffer. The indices loop around back to the start of the array once they exceed the maximum size.

`Getc` and `Putc`

`Getc` and `Putc` simply send a message to the identified tid which has a source indicating that it comes from a user task, and the given `char` (for `Putc`). `Getc` does not send a `char` to the server but does return the character that the server replies with.

Train Control

Command Processing

The commands are read in, one character at a time by the command server. The command server prints out the characters that are being entered as they are being entered and stores the entered characters in a command buffer. When the the “enter” character (13) is read in, the terminal scrolls up, the command is processed, and the buffer is reset. Command processing consists of a number of checks to determine if the command is a valid command or not, and then acting on the command if it is valid. If the command is not valid, `Invalid Command` is printed on the terminal in the command section to alert the user. If the command is valid it is sent directly from this task to the train control through `Putc`.

One downside of this approach is that when command execution requires delays (as in reverse or switch), the user input and terminal also appear to freeze.

Sensor Readings

In a separate process from the command server, the sensors command is given and then the sensors are read, processed, and printed. A byte from the Marklin set is read it, and then processed. As in command processing, there are no major data structures required here, as each byte is processed once and then discarded. If a sensor has been triggered, then it is printed on the terminal, and the index of the location of the terminal where it belongs is incremented so the next sensor that is triggered will be printed in the next location on the list.

User Interface

UI Server

The UI server acts as an interface between the tasks that may want to print to the terminal and the terminal output task (`Putc`). It’s primary purpose and function at the moment is to buffer strings that it receives from the tasks and send them to the terminal through `Putc`. Currently only this task sends anything to the terminal and all other tasks must go through it. This ensures that output from different tasks does not end up interleaved and garbled between them. It is the responsibility of the sending tasks to ensure that any string sent to this server is complete and that no problems will arise if another task sends something between two sent strings.

Main Event Loop

Immediately after setup is finished, the main event loop of the kernel begins. This loop continues to execute until there are no tasks in the ready queue to be run, at which time the program will exit. The code for this loop can be found in `src/kernel/kernel.c` in `k_main()`.
The loop calls the scheduler to get the next task to be run, runs that task until an interrupt is triggered, and then acts on the result code that is passed from that task. That code can tell the kernel to create a new task, return the just-run task's id, its parent's id, or exit that task. The kernel can also pass data back to the task by overwriting the location on the user's stack that will be re-loaded into `r0` the next time that task is scheduled.

Task Descriptors

The task descriptors are allocated within their own memory location in the kernel space. The task descriptor of a task can be found using the id of that task by taking that base memory location and then subtracting the size of the task descriptor times the id of the task plus one. In this way, the task descriptors behave like an explicitly-managed array of arbitrary (to a point) size.

Scheduling

Tasks are scheduled by their priority and then by the least recently run, with newly-created child tasks of the same priority as the parent scheduled before the parent next runs.
At the moment, scheduling uses one "priority queue" to manage the tasks that are in the ready state. Although no other queues are used at the moment, the same system will be easy to apply to them as well once they're needed in the next kernel assignment.
The ready queue is modeled as a binary heap, where the root node is the next task that will be scheduled. This heap is implemented as an array where a node at index `i` has child nodes at indices `i*2` and `i*2+1`, and functions are implemented to keep the heap sorted when adding or removing nodes.
When the kernel needs to schedule a new task, it removes the root node from the ready heap, and then resorts the heap so the next task to run becomes the new root node. Then, after the task runs, it is put back onto the ready queue, and the heap again ensures that it is sorted correctly.

Context Switch

There are two context switch functions, both of which are defined in `src/kernel/swi.S` and both are written in assembly. One, `enter_kernel`, is used to enter kernel mode, and the other,

`leave_kernel`, is used to leave kernel mode (a true triumph of naming!). They are mirror opposites of each other.
The user process enters the kernel by calling `enter_swi` with one argument (`&args`) which is the address of the struct `Args` which stores all the parameters needed for the kernel to determine the call type, and is stored in `r0` when entering the kernel mode. This is the data structure that is used for passing information from the user tasks to the kernel. Through the context switching process, we ensure that the register holding `Args` never gets overwritten at any point so that it can be successfully transferred between the tasks and the kernel. Upon leaving the kernel, we overwrite the value in the user stack that will later be reloaded back to `r0` as the return value to the user task.
When entering kernel mode, supervisor mode is switched to, then the current registers are stored on the user stack, the values that were stored on the kernel stack are loaded into the registers, and the kernel resumes execution where it left off.
When leaving the kernel, the current registers are stored on the kernel stack, the values that were stored on the user stack are loaded into the registers, user mode is switched back into, and the user task resumes from where it left off.
User tasks can trigger a software interrupt by calling `enter_swi,` then entering kernel which is defined as triggering the `enter_kernel` function, which switches to kernel execution. And the kernel hands off to a user task by calling `leave_kernel` directly.

Send-Receive-Reply

In the send-receive-reply structure, two tasks communicate with each other through a three-stage process. The sending task provides the id of the task it wants to communicate with, the message it wants to send, and a buffer in which to receive the reply. The receiving task provides a buffer in which to receive a message, and once a task has sent to it, it processes the message that is put into the receive buffer. Once it has the information to pass back to the task that the task was waiting for, it replies to that task with a reply message, which is inserted into the sending task's reply buffer, and the sending task is unblocked.

Data Structures and Storage

Because there is so much data that needs to be passed between tasks, and tasks that can be blocked for an indefinite amount of time, during which the data needs to be stored in the kernel, there are a number of techniques used to manage the tasks and the data that belong to those tasks during the message passing process.
The messages, length information, and buffers of blocked tasks are stored within a struct in that task's task descriptor, and can be accessed at any point in the future as long as the task id in question is known. A strictly-FIFO queue is in the task descriptor of any receiver task, which holds the task ids of all tasks waiting to send to it. Any receiving task also manages an unordered list of any tasks that have sent to it and are waiting for a reply. This list exists solely

so that the message passing process can be aborted properly if the receiving task exits before replying and closing the loop.

Send

When a task wishes to send a message, it passes the id of the task it wants to send to, a message, and a reply buffer into the kernel. The pointer to the reply buffer is stored in the sending task's task descriptor. Then the kernel checks to see if the task identified as the receiver is `RECEIVE_BLOCKED` or not. If that task is `RECEIVED_BLOCKED`, it immediately passes the message along to the receiver, moves the receiver back into the ready state and onto the ready queue, and marks the sending task as `REPLY_BLOCKED`. If the receiving task is not `RECEIVE_BLOCKED`, the message is stored in the sending task's task descriptor, the sending task id is added to the receiver's send queue, and the sending task is set to the `SEND_BLOCKED` and will wait for the receiving task to pull it from its send queue.

Receive

The process of receiving is a mirror of the process of sending. When a task wishes to receive a message it passes a receive buffer and a pointer to an int on its stack. When it gets to the kernel, if there is a task on its send queue, the message from that task is put into the send buffer, the send task is set as `REPLY_BLOCKED`, and the receiving task is put back onto the ready queue.
If the send queue is empty, then the buffer and int pointer are put into the receiving task's task descriptor, its status is set to `RECEIVE_BLOCKED`, and it waits for a task to want to send a message to it.

Reply

When a receiving task has received and processed a message from a sender and is ready to reply with the results, it calls `Reply` and passes it the id of the task to reply to, and the message to send to that same task. The kernel then copies that message into the sending task's reply buffer. Then both tasks' states are set to `READY` and they are put back onto the ready queue.

Name Server

The name server initializes itself, create hash table with dynamic bucket size to store name to id mapping using the heap described below. It then calls Receive, blocking itself until a task wants to register with it or request an id. Tasks register with it by calling `RegisterAs`, which sends the task name and id to the server. A task can request a task's id by calling `WhoIs` with the name of the task whose id it needs, which also initiates a send-receive-reply sequence. The name server keeps track of the names and ids it has received through a hashmap, which facilitates fast lookup and reduces the space required to store the name-id matches.

Heap

There are often times we want an arbitrarily-sized container to hold data that can live outside the scope of stack and can be destroyed and created as needed. All information about the heap remains within the kernel state struct, and is allocated and freed by the kernel. When we initialize the kernel, we allocate `0x300000` byes for the heap space which starts from `0x1F00000` and goes to `0x1C00000`. This memory space is further divided into three block groups for finer grained management and to prevent internal fragmentation.
Each block allocates memory of different sizes: `64` bytes, `256` bytes and `1024` bytes. The actual user usable amount is `16` bytes less which is used for metadata. Each time a user requests heap space, with a size provided, for some size space, they receive a block of the smallest size that works.
A memory allocator to manages the block heap. The strategy is that we have six looped linked list in total. And each entry in three of the linked list holds an empty block that a user can use and its metadata. The other linked list is comprised of the blocks that are currently in use. They are called `heap_block_used` and `heap_block_unused` respectively, and each block size has its own set of circular linked lists. Blocks are moved from the unused list to the used list when they are allocated, and then back to the unused list when freed, which recycles the memory block for the next allocation.
The kernel maintains information about the state of the heap, which includes the number of used blocks. One downside of this approach is that even if a user requests 4 bytes, they are allocated a 64 byte block. This is unavoidable with this heap implementation strategy.
Because the kernel manages everything about heap, `Malloc` and `Free` are system calls which switch the execution into kernel mode to request heap space. These are implemented in the same manner as the other kernel system calls for message passing and task management.

Clock Server

Execution

Similarly to the Name Server implemented in the last assignment, the Clock Server executes within an infinite while loop. On each iteration of the loop, it calls `Receive()` and waits for a task to want to send a message to it. After receiving a message, it determines what type the message is, and then executes accordingly. If the message is a `TIME` request, it simply replies with the value of its internal ticks counter. If the message is a `DELAY` request, it calculates the tick value when that task should be replied to, and then adds the task id to the queue of delayed tasks. It does the same for a `DELAY_UNTIL` request, except that instead of calculating the delayed tick value, it checks that the delay time is greater than the current tick count, and then sets the requested time as the tick value, and adds the task id to the delayed tasks queue. Finally, if the message is a `TICK` request, then it is from the clock notifier, we then update the value of the internal ticks, and reply back to the clock notifier to unblock it. We release any tasks that are on the delayed priority queue waiting for this particular time, sending them the current number of ticks. The server expects that only the Clock Notifier will send `TICK` requests.

Data Structures

Clock Messages

Clock messages are sent as a struct with two integers: the message type, and a time field. The message type is an enum, which defines the type of message. This is the field that that clock server checks to determine how it needs to respond to the message. The time field is used both to send tick amounts to the clock server, and for the server to send the current times back to the sending tasks.

Min Heap

The tasks that have been delayed and are waiting for their ticks to be up are stored in a min-heap, which is implemented as two arrays, one which orders the task ids as a min-heap (sorted by their wake time), with the children of the task at index i stored in `i*2` and `i*2+1`, and the other of which is a mapping from the ids to their wake times. The `add` and `remove` methods automatically keep the minheap sorted. We generalize priority queue from k1 by adding function pointers to make it more flexible. This implementation was already discussed in k1 as our implementation of the ready queue, so we won't go into it in detail here.

Clock Notifier

The clock notifier is a very simple task, started by the clock server, which simply loops endlessly, first waiting for a timer event, which is set up (in the kernel initialization) to trigger an interrupt every 10ms, and then sending a message to the clock server to inform it of the update.

Interrupt Handler

The interrupt handler checks that the interrupt that has been triggered is a timing interrupt; if it isn't then it returns. This can easily be expanded to include more interrupt types and checks as more interrupts are needed. For this assignment we only need the timing interrupts. Then, if the interrupt is for a timing event, all tasks that are waiting on the `TIMER_EVENT` queue are woken up and put back on the ready queue. Finally the timer interrupt is cleared.
The interrupt handler registration process during the setup is similar to that of the swi handler except that the memory location is `0x38` instead of `0x28`. The details of entering interrupt can be found in `src/kernel/swi.S`, `enter_interrupt` section. When entering the interrupt assembly routine, we store the user registers first by going into system mode, we then switch interrupt mode to save the `lr`/return address. An important thing here is to get `lr` by `lr - 4`

instead of lr itself because of the pipelining effect of architecture. And we also have to store `spsr_usr` and `lr_usr`. And go back to supervisor mode and change `pc` to enter kernel.
Tasks that want to wait for an event, can call `AwaitEvent()`, which enters kernel mode, and the kernel puts the task into the `EVENT_WAIT` state and adds them to the queue of the event they wish to wait for. If the event they are waiting for is unrecognized, the task is placed back on the ready queue with an error code as its return value.

Idle Task

The idle task sets itself up by first unlocking the `DeviceCfg` location (`0x80930080`) by setting a flag in the `SysSWLock` location (`0x809300C0`), and then setting a flag in the DeviceCfg location which allows the `Halt` location (`0x80930008`) to be read. Once those are set up, the idle task enters an infinite while loop which simply reads from the `Halt` memory location once per loop. Reading from this location puts the CPU into low power mode until an interrupt is triggered.
The idle percentage counter computes the percentage of time that the CPU spends within the idle task (halted) during the execution of the program, and outputs it to the terminal. In the main kernel loop, immediately before a task is switched to and immediately after a task is switched from, the `task_performance` function is called. This function determines how much time it has been since the function was last called, and then adds that to the `total_ticks` counter. If the most recent run task was the idle task, it also adds that time to the `idle_ticks` counter. It then computes and stores the tick information of the idle task and kernel. The percentage of time that the program has spent in the idle state is computed by dividing the `idle_ticks` by the `total_ticks.`