AbyssOS: TC1 Writeup

1 Summary

This is a writeup for Train Control Assignment 1 of CS452, where we have added in a number of train controlling commands and functionality to support directing a single train towards a location on the track, then stopping within a short distance of that location (plus or minus some offset). To facilitate these commands, the UI of the terminal has also been upgraded to support new commands for calibration, localization, movement and looping, as well as displaying train statuses and logging in the terminal.

2 Kernel Structure

2.1 K1 Contributions

The main component of our kernel is the `Kernel` class:

```
class Kernel {
public:
    Kernel();
    ~Kernel();
    void schedule_next_task();
    void activate();
    void handle();
    void handle_syscall();
    void handle_interrupt(InterruptCode icode);
    void start_timer();

private:
    int p_id_counter = 0;
    int active_task = 0;
    InterruptFrame* active_request = nullptr;

    Task::Scheduler scheduler;
    Descriptor::TaskDescriptor* tasks[Task::USER_TASK_LIMIT] = { nullptr };
    SlabAllocator<Descriptor::TaskDescriptor, int, int, int, void (*)()> task_allocator
     = SlabAllocator<Descriptor::TaskDescriptor, int, int, int, void (*)()>(
       (char*)Task::USER_TASK_START_ADDRESS, Task::USER_TASK_LIMIT);
    Clock::TimeKeeper time_keeper = Clock::TimeKeeper();

    // clock notifier "list", a pointer to the notifier
    int clock_notifier_tid = Task::CLOCK_QUEUE_EMPTY;

    void allocate_new_task(int parent_id, int priority, void (*pc)());
    void handle_send();
    void handle_receive();
    void handle_reply();
    void handle_await_event(int eventId);

    int idle_tid = SystemTask::IDLE_TID;
};
```

This class wraps the kernel functionality and data structures, so it's a useful reference point for talking about structure and functionality. Most of the class contents are straightforward or self-explanatory, so we will only cover the components that hold nontrivial complexity.

Scheduling: To schedule the order of user tasks, the kernel defers to the `Scheduler`, with `schedule_next_task()` also making use of the class functionality:

```
#define NUM_PRIORITIES 8

class Scheduler
{
    public:
        Scheduler();
        int get_next();
        void add_task(int priority, int task_id);

    private:
        RingBuffer<int> ready_queue[NUM_PRIORITIES];
};
```

The scheduler itself is very simple, being little more than a wrapper around 3 stack-allocated ring buffers (the ring buffers themselves are a template class with a fixed size of 512 slots; 512 was chosen because we reasoned this would be large enough for any use case). Each buffer stores task IDs and corresponds to a different priority level – high, medium and low – where each queue can be added to on request, and `get_next()` will loop through queues in order to determine which queue should be popped from when called. 8 priorities was determined to be sufficient for our needs (as of K3).

Task Storage: The kernel keeps track of a SlabAllocator structure, which keeps track of slab allocations. It is a templated class with variadic arguments, which allows it to construct arbitrary classes in slabs, and it internally uses a ring buffer to keep track of free memory locations. For user task descriptors, slab allocation is set to start at the hardcoded address of 0x10000000, which we determined to be a safe address for slab allocations (will not collide with important memory addresses).

```
template <typename T, typename... Args>
class SlabAllocator {
public:
    SlabAllocator(char* starting_location, int total_slabs);
    ~SlabAllocator();
    T* get(Args... arguments);
    void del(T* target);
    int get_remaining_size();

private:
    int size;
    int T_size;
    RingBuffer<char*> slabs;
};
```

The slab allocator maintains basic free and delete functions, allowing it to perform similar functionality to that of a heap.

Task descriptors store the task ID, task parent ID, the stack of the task that they represent and some other useful status information.

Activate/Handle: These functions are called in our `kmain` function, which schedules tasks and calls these two functions in an infinite loop. `activate()` initializes user tasks if they have not been initialized and context switches into them, while `handle()` responds to system calls using the kernel-defined `HandlerCodes`.

2.2 K2 Contributions

The main architecture contributions of K2 are task state and message passing functionality. To support this, task descriptors have been supplemented with some new functions and data structures:

```
class TaskDescriptor {
public:
    enum TaskState { ERROR = 0, ACTIVE = 1, READY = 2, ZOMBIE = 3, SEND_BLOCK = 4, ... };
    TaskDescriptor(int id, int parent_id, int priority, void (*pc)());
    // message related api
    void queue_message(int from, char* msg, int message_length); // queue_up a message
    bool have_message();
    int fill_message(Message msg, int* from, char* msg_container, int msglen);
    int fill_response(int from, char* msg, int msglen);
    Message pop_inbox();

    InterruptFrame* to_active();
    void to_ready(int system_response, Scheduler* scheduler);
    bool kill();
    void to_send_block(char* reply, int replylen);
    void to_receive_block(int* from, char* msg, int msglen);
    void to_reply_block();
    void to_reply_block(char* reply, int replylen);

    bool is_active();
    bool is_ready();
    bool is_zombie();
    bool is_send_block();
    bool is_receive_block();
    bool is_reply_block();

    const int task_id;
    const int parent_id; // id = -1 means no parent

    friend class Kernel;
protected:
    void show_info();

private:
    TaskState state;
    int priority;
    int system_call_result;
    bool initialized;
    void (*pc)();
    MessageReceiver response;
    RingBuffer<Message> inbox;
    char* sp;
    char* kernel_stack[USER_STACK_SIZE];
};
```

These different functions can place tasks into different states, which may or may not remove them from scheduling or flag them as having passed into different states of message passing. Tasks also now maintain a message inbox, which allows them to receive messages even if they aren't actively waiting for them.

As of K2, `USER_STACK_SIZE` is equal to 131,072, equivalent to 128kB of stack space. This was raised up from 16kB (previously 4kB) to make room for an unordered map for the name server.

The message passing functions themselves, then, are responsible for putting tasks into different states, as well as performing memcpy on the messages to transfer them to the right spots. Here's the `handle_send()` function, as an example.

```
void Kernel::handle_send() {
    int rid = active_request->x1;
    if (tasks[rid] == nullptr) {
        // communicating a non existing task
        tasks[active_task]->to_ready(NO_SUCH_TASK, &scheduler);
    } else {
        char* msg = (char*)active_request->x2;
        int msglen = active_request->x3;
        char* reply = (char*)active_request->x4;
        int replylen = active_request->x5;
        if (tasks[rid]->is_receive_block()) {
            tasks[rid]->fill_response(active_task, msg, msglen);
            tasks[rid]->to_ready(msglen, &scheduler);
            tasks[active_task]->to_reply_block(reply, replylen);
        } else {
            // reader is not ready to read we just push it to its inbox
            tasks[rid]->queue_message(active_task, msg, msglen);
            tasks[active_task]->to_send_block(reply, replylen);
        }
    }
}
```

Beyond this, we also create two servers using these message passing functions: the name server and the Rock/Paper/Scissors server. Both servers wait for send requests in an infinite loop, and process decisions only when their receive calls are fulfilled.

- The name server handles name registration and name lookup requests through the `WhoIs()` and `RegisterAs()` system calls. It uses a templated hashmap to keep track of names and task IDs, where names are mandated to be at most 16 characters in length (longer names are truncated). Duplicate names in `RegisterAs()` overwrite the the name server's mapping, while unregistered names in `WhoIs()` cause it to return a pre-specified error code.

- The Rock/Paper/Scissors server allows special clients to sign up for and play in Rock/Paper/Scissors matches against each other by sending special messages to the server. It mediates interactions between clients, keeps track of matches and results and even returns some output showing the results. This will be further discussed in the Kernel Output section.

The templated hashmap used by the name server uses a generic FNV hash function with separate chaining to resolve collisions, and is courtesy of the Embedded Template Library for C++.

2.3 K3 Contributions

K3 adds interrupt handling, clock functionality and some client and idle tasks, which test these functionalities and track how much time the kernel is spending idle.

Interrupt Handling: Besides the assembly-based interrupt handler, managing interrupts also requires slightly different user re-entry due to the volatility of interrupts. Interruption status is tracked via a bool in the task descriptor, and when returning to user tasks, this bool is used to determine if we want syscall re-entry (optimized for ABI rules) or interrupt re-entry (assume nothing).

Clock Functionality: To keep track of the clock, clock interrupts, clock ticks, idle time calculations and clock system calls, we introduce the `TimeKeeper` class along with two user tasks: the clock server and the clock notifier. Upon initialization, the kernel enables interrupts, then instantiates an instance of the `TimeKeeper` class, which activates the compare registers and enforces that they occur every 10ms.

- The clock server is responsible for handling the three clock-related system calls: `Time()`, `Delay()` and `DelayUntil()`. Upon receiving a request, the clock server will either reply with an internally-tracked tick integer or add the task ID to an internally held priority queue (of maximum size 64, which is enough to keep track of hopefully enough tasks but not so many as to cause excessive cache misses), which is checked every tick to determine if there is a task that needs to be awoken from delay.
- The clock notifier is awoken whenever a timer interrupt is fired, which is handled using `AwaitEvent`. After being woken up, the clock notifier sends a message to the clock server telling it a tick has occurred, then calls `AwaitEvent` again.
- Here's what the TimeKeeper looks like:

```
class TimeKeeper {
public:
    TimeKeeper();
    ~TimeKeeper();

    void start();
    void tick();
    void calculate_and_print_idle_time(int active, int prev, int idle);

private:
    void set_comparator(uint32_t interrupt_time, uint32_t reg_num = 1);
    uint64_t tick_tracker = 0;

    // Time tracking variables
    uint64_t idle_time = 0;
    uint64_t last_ping = 0;
    uint64_t total_time = 1; // start at 1 to avoid division by 0 errors
    uint64_t last_print = 0;
};
```

  Here `start()` begins timer tracking, `tick()` advances the internal clock by one tick (10ms) and resets the comparator, and `update_time()` updates idle time calculations (and possibly prints the results to the terminal).

- Client and idle tasks: they work pretty much as specified in the K3 assignment spec. Clients ask the first user task for a given delay and repeat amounts, then print whenever their delays expire. The idle task, which has a lower priority than all clients, simply yields in a loop.

2.4 K4 Contributions

The K4 contributions are almost all users tasks, which we will take about later. However, we of course now have the additional complications of the UART, its associated interrupts, and the CTS signals.

In-order to enable interrupts, we permanently open IRQ 145 which corresponds to GPIO 24, where both UART channels are able to communicate with us through interrupts. This naturally causes a problem: every UART interrupt imaginable comes from the same pipeline, 145, thus, we need to check a special register, `IER` to ensure everything is cleared. If we are ever interrupted through IRQ 145, we will continuously read and clear / disable interrupts as indicated by the `IER` register until both UART0 and UART1's `IER`s return `0x1`, which indicate that all interrupts are cleared.

As we may have mentioned, sometimes we clear interrupt, sometimes we disable interrupt. define **clearing** as the action disable the interrupt naturally, while **disable** as the action to tell `IER` registers to disable certain interrupts. Both methods have upside and downside. The **clearing** approach ensures that you are interrupted as soon as possible, and avoid missing potential interrupt ticks and fall behind. For example, we relies on `MSR` register and the modem interrupt to read and handle `CTS` interrupt, which calls us every time the level of `CTS` changes. This action need to be handled urgently, since going up (cannot write) and going back down (can write again) may arrive very closely to each other, the risk of turning off the interrupt and potentially miss a tick is just not acceptable. (this can permanently killed the UART1 server until we introduce timeout). However, this approach is not universally viable, since sometimes you have to disable interrupts since there is no way to clear it. For example, the `THR`, which indicate the transmit interrupt cannot be cleared unless you fill the buffer above a certain threshold again. This is either difficult or impossible to do, thus, it is better to shut them down completely during kernel and just let the corresponding listeners transfer the message to kernel.

However, it is pretty good that most interrupt we have to handle either won't have a risk of skipping if we disable, or have a risk of skipping but can be handled by simply clearing the interrupt.

It also worth mentioning that we are trying our RTOS as a hard RTOS, meaning skipping an interrupt is unacceptable behaviour and is considered as system failure. So for, to listen to each interrupt, we only have 1 listener for each, but our code is fast enough such that it totally fine (like, we are at 98% idle time)

Following is the core of how we handle uart interrupt, though not pretty, it works pretty well. (We will be extracting this feature to a specific subclass that records and handles the interrupt). The text is pretty small, so I recommend reading the actual branch if you are interested in more details.

```
case InterruptCode::UART: {
    /**
     * Note that no matter which interrupt, you receive from the same id, UART_INTERRUPT_ID
     * this is kinda a problem, since a large variety of stuff can be from that type of interrupt,
     * even same type, but uart0 vs uart1
     *
     * in order to differentiate them, we relies on checking the register later, IIR,
     * IIR includes both the information about if there is an interrupt to handle, and if so, what is the interrupt exactly.
     *
     * the general work flow is 1. we receive UART_INTERRUPT_ID interrupt, thus recognize interrupt happened
     * we check IIR to see what type of interrupt happened, we could potentially get a large quantity of interrupt
     * overall, the goal is that if we receive interrupt in the form of UART_INTERRUPT_ID, we keep cleraing interrupt
     * until every interrupt associated with uart is cleared
     *
     * Also note that server is in control of which register is flipped, thus also in control of which interrupt is happening.
     */

    int exception_code = (int)(uart_get(0, 0, UART_IIR) & 0x3F);
    do {
        if (exception_code == UART::InterruptType::UART_RX_TIMEOUT && uart_0_receive_tid != Task::UART_RECEIVE_EMPTY) {
            int input_len = uart_get_all(0, 0, tasks[uart_0_receive_tid]->get_event_buffer());
            tasks[uart_0_receive_tid]->to_ready(input_len, &scheduler);
            uart_0_receive_tid = Task::UART_RECEIVE_EMPTY;
            enable_receive_interrupt[0] = false;
            interrupt_control(0);
        } else if (exception_code == UART::InterruptType::UART_TXR_INTERRUPT && uart_0_transmit_tid != Task::UART_TRANSMIT_FULL) {
            tasks[uart_0_transmit_tid]->to_ready(0x0, &scheduler);
            uart_0_transmit_tid = Task::UART_TRANSMIT_FULL;
            enable_transmit_interrupt[0] = false;
            interrupt_control(0);
        } else if (exception_code == UART::InterruptType::UART_CLEAR) {
            break;
        } else {
            printf("Uart 0 Too Slow \r\nexception code: %d receive_tid: %d transmit_tid %d\r\n", exception_code, uart_0_receive_tid, uart_0_transmit_tid);
            while (true) {
            }
        }
        exception_code = (int)(uart_get(0, 0, UART_IIR) & 0x3F);
    } while (exception_code != UART::InterruptType::UART_CLEAR);

    exception_code = (int)(uart_get(0, 1, UART_IIR) & 0x3F);
    do {
        // this is a really shitty way to handle this, I think it would probably be better if we something similar to a dedicated class object
        // but we will fix it soon once experiementa go through.
        if (exception_code == UART::InterruptType::UART_RX_TIMEOUT && uart_1_receive_timeout_tid != Task::UART_RECEIVE_EMPTY) {
            tasks[uart_1_receive_timeout_tid]->to_ready(0x0, &scheduler);
            uart_1_receive_timeout_tid = Task::UART_RECEIVE_EMPTY;
            enable_receive_interrupt[1] = false;
            interrupt_control(1);
        } else if (exception_code == UART::InterruptType::UART_RX_INTERRUPT && uart_1_receive_tid != Task::UART_RECEIVE_EMPTY) {
            tasks[uart_1_receive_tid]->to_ready(0x0, &scheduler);
            uart_1_receive_tid = Task::UART_RECEIVE_EMPTY;
            enable_receive_interrupt[1] = false;
            interrupt_control(1);
        } else if (exception_code == UART::InterruptType::UART_MODEM_INTERRUPT && uart_1_msr_tid != Task::UART_TRANSMIT_FULL) {
            char state = uart_get(0, 1, UART_MSR);
            if ((state & 0x1) == 0x1) {
                tasks[uart_1_msr_tid]->to_ready(0x0, &scheduler);
                uart_1_msr_tid = Task::UART_TRANSMIT_FULL;
            }
        } else if (exception_code == UART::InterruptType::UART_TXR_INTERRUPT && uart_1_transmit_tid != Task::UART_TRANSMIT_FULL) {
            tasks[uart_1_transmit_tid]->to_ready(0x0, &scheduler);
            uart_1_transmit_tid = Task::UART_TRANSMIT_FULL;
            enable_transmit_interrupt[1] = false;
            interrupt_control(1);
        } else if (exception_code == UART::InterruptType::UART_CLEAR) {
            break;
        } else {
            while (true) {
            }
        }
        exception_code = (int)(uart_get(0, 1, UART_IIR) & 0x3F);
    } while (exception_code != UART::InterruptType::UART_CLEAR);
    UART::clear_uart_interrupt();
    break;
}
```

2.5 Context Switching

Most of our context switching procedure is relatively standard/inflexible, so I'll just jot down some key features here:

- For the sake of optimization, we maximally assume ABI rules hold when doing system call context switching. Indeed, we can even disregard the value of the program counter, since calling the system call means that the link register now holds the correct value to return to. On the other hand, the interrupt handler saves and loads all registers when entering/exiting the kernel, and also does a bit of register juggling to keep track of the program counter and stack pointer.
- When the context switch returns to the kernel, it passes the kernel a pointer to an `InterruptFrame` struct, which is essentially a format of the 31 main non-zero registers, allowing easy access, along with 3 extra 8-byte chunks of information: the program state of the user, read from SPSR_EL1; the program counter of the user, read from ELR_EL1; and a data chunk, which indicates that the user task was interrupted rather than performing a system call.
- The first time a user task is switched into is different from subsequent switchings, as registers do not need to be loaded the first time.
- The x0 register is used liberally to pass around arguments and return values.

3 User Tasks

3.1 Initialization Scheme

As of K3, we are responsible for setting up the following (increasingly complex) user task scheme:

1. Upon initialization, the kernel launches with one task in its scheduler: the launch task (located in `src/user/user_tasks.cc`), which creates the following user tasks in order:
   - The name server, with priority 2.
   - The clock server and clock notifier, each with priority 1.
   - The idle task, with priority 4.
   - The four UART servers, each with priority 1 (and each of which creates its own notifier.
   - The global pathing server (aka train engineer, aka train proprietor) with priority 2.
   - The local pathing servers (each responsible for one train) and each with priority 2.
   - The train and sensor admin servers, each with priority 2.
   - The terminal admin, with priority 3.
2. The name server keeps an unordered map that keeps track of names and tids corresponding to those names, and responds to `RegisterAs` and `WhoIs` requests.
3. The clock server and clock notifier keep track of 10ms clock ticks and respond to `Time`, `Delay` and `DelayUntil` requests (more on the clock above).
4. The UART servers and notifiers keep track of I/O through UART interrupts, and respond to `Getc`, `Putc`, `UARTReadRegister` and `UARTWriteRegister` requests.
5. The train admin is responsible for sending and coordinating commands to the Märklin track through the UART1 sender server. In particular, switches and reversing commands, which require delays before subsequent commands, have their complexity handled here.
6. The sensor admin is responsible for regularly querying for and collecting sensor data, and maintaining a subscriber system that responds to queued tasks that are waiting on sensor data.
7. The terminal admin is responsible for sending and coordinating signals to the terminal through the UART0 sender server. Importantly, it controls command prompt functionality, switch and train status printouts, timer printouts, idle time printouts, and debug printouts.
8. The idle task sits in a loop and waits for interrupts, and is the task that the kernel will default to when no other tasks are scheduled. Alongside the idle time task is a special terminal courier, which tells the terminal to ask the kernel for idle statistics every two seconds. These idle stats are then turned into a percentage and printed on the terminal display.

3.2 RPS Server/Clients (K2)

The RPS Server, like the name server, handles sends/receives from RPS clients, coordinates match logic and keeps track of matches (storing matches in a large, linearly-searched array).

Clients loop through the following logic loop:

1. Ask for the task ID of the RPS server, and send it a signup request.
2. Query the current system time to make a "random" play decision, and send it to the server.
3. Repeat the previous step $3 + \text{randint}(0, 2)$ times, then send a quit message.
4. After sending a quit message, or after receiving a quit message from the server, either destroy self (75% chance) or rejoin the match pool (25% chance).

3.3 I/O Servers (K4)

The I/O management of our system can be broken down into three main components:

3.3.1 UART Interfacing

The Objective of UART interfacing is to avoid busy waiting as much as possible, a.k.a we only read if hardware tell us data is available (as well as we want to read). The work done requires working with very low-level register primitives, talking directly to the UART cable and making use of general-purpose IO (GPIO) and special UART interrupt functionality to allow the kernel to manage I/O in an interrupt-based fashion.

We chose to make use of a four-server architecture, one for each of the combinations of UART channel 0/1 and input/output, in the hopes that this would lead to optimal overall performance due to servers not needing to block on each other's workflows.

Notably, this particular architecture choice takes advantage of the fact that each of the UART use cases is relatively specialized. UART0 input and output are used exclusively by terminal tasks, which in our system we have confined to one task each, so the server usage can be tightly controlled. UART1 input is also used exclusively for reading sensor data, so specializing the server for that purpose – specifically, the purpose of reading 10 bytes every 100ms (in our system) – makes for a more optimized structure overall. UART1 output, which controls all train command functionality, requires a bit more care, although setting it to its own server gives us more modularization and control regardless.

As for the train and sensor, we decided to split them apart. Train server mostly handles the train command and the track command which are mostly input, while the sensor server is mostly reading output from uart1. this splitting of workflow allow an almost complete decouple of workflow between input and output, while allowing any server that relies on these server to talking to one of them have no influence on the other (except on the wire physically but is outside the scope of K4 for this case).

The Train server either pump corresponding byte right away to the uart1 write server, or create a task that pump some control byte to the uart1 server after some delay. However, there is no much write protection right now (for example, train is allow to actually write speed command which will be overwrite by reverse command's later acceleration after delay), but they will be enfored in TC1.

The Sensor server have a worker that just keeps pulling from uart1 read server with a 100 ms delay (though we are not sure if we even need it cause MMU and data cache actually makes our code go flying, only concern is the 2400 baus rate and the fact that the wire is 1 directional). Any task can subscribe to the sensor server, which will be updated as soon as the new reading comes in. The workflow is pretty much (Sensor worker detect update $\longrightarrow$ notify Sensor admin $\longrightarrow$ notify all subscriber $\longrightarrow$ Sensor worker listen for update). This allow the most up to date information for all subscribers, and if any subscriber want continuous update, it should re-subscribe right after getting back as well.

3.3.2 Terminal management

The ultimate end-user functionality requirement of K4 is to have recreated the functionality of A0, which necessarily requires a considerable amount of terminal management. Specifically, we need to gather and display information about system time, idle time, sensors and switches, and also allow users to control train speeds, train directions and switch layouts through a command line interface. To recreate this, a dedicated user input "courier" is used, which is the only task that reads from UART0, and a dedicated terminal server is used, which is the only task that is allowed to write to UART0. All UART0 writing requests must go through the terminal server, which carefully controls and formats output.

3.3.3 User task architecture

On a more general level, the layout of the user tasks as a whole has been carefully designed and planned out to weave a complicated net of tasks, making heavy use of the ideas of servers, couriers and workers to offload work to other tasks if necessary and make sure nothing is needlessly blocked. Here's a rough sketch of what the current system architecture looks like:

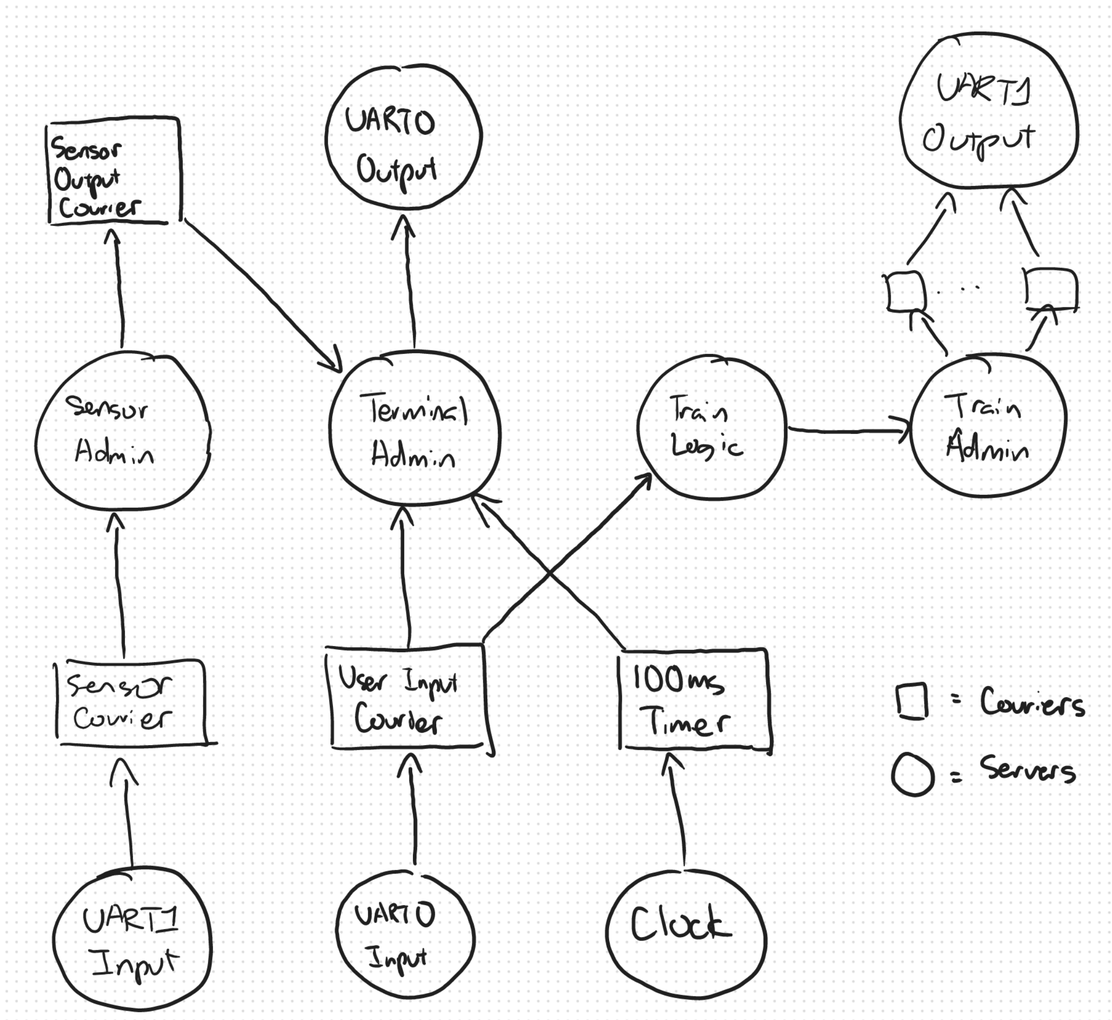

Figure 1: A highlighted subset of our K4 architecture additions. Not shown: the idle task/idle timer task.

In general, couriers are used when information needs to be passed between tasks without blocking receiving. The train admin server in particular makes use of a courier pool to service different train commands without becoming blocked itself, especially as some train commands may be quite slow to process due to the slow speed of train command listening and/or UART1 transmission.

3.4 Train Control P1 (TC1)

To handle our TC1 train control system, we introduce the global pathing server, the local pathing workers, and the `TrainStatus` class.

3.4.1 Local Pathing Workers

When trains receive commands, our train control system may need to lock out further commands from being issued to those trains. However, this lockout system must not block any important servers, such as the terminal admin and the global pathing server (the train engineer/proprietor). As such, we create a system of "local" pathing workers, which are set up to receive commands from the terminal admin, process those commands, send translated commands to the global pathing server, and sleep until they are replied to (and as a result, refuse further commands). There is one local pathing worker for each train.

3.4.2 The `TrainStatus` Class

To keep track of train status, we keep a per-train class that tracks every single thing related to train controlling, including:

- Information that tracks the position, velocity and acceleration of trains.
- Information that relates to the current path of the train and expected sensor hits.
- Information gathered from train calibration processes.

Alongside all this information is a mountain of functionality:

- Functions for locating and moving the train.
- Functions that launch train calibration procedures.
- Functions that subscribe the train to certain sensors, or unsubscribe.
- Functions for manipulating the state machine of the train.
- Functions for entering and exiting a loop on the track.
- Functions for starting and stopping the train on the track.
- Functions for handling the results of sensor notifications, for all different current train states.

Here's a heavily abbreviated code sample, which shows the most important functions:

```
class TrainStatus {
public:
    void locate();
    void pre_compute_path(bool set_switches = true);
    bool goTo(Dijkstra& dijkstra, int dest, SpeedLevel speed);
    bool calibrate_velocity(bool from_up, int from, SpeedLevel speed);
    void enter_loop(SpeedLevel speed);

    void subscribe(int from);
    void sensor_notify(int sensor_index);
    void continuous_localization(int sensor_index);
    void clear_traveled_sensor(int sensor_index);
    void handle(int sensor_index);
private:
    bool toSpeed(SpeedLevel s);
    void pipe_tr();
    void refill_loop_path();
};
```

3.4.3 The Global Pathing Server

The train brain, the train engineer, the train proprietor. It has many names, because it's very important an encapsulates a ton of functionality. For us, this is known as the global pathing server, and it is where commands from local pathing workers are received and processed, and from where commands to make trains move in coordinated ways are issued.

Insofar as TC1 is concerned, the global pathing server is only responsible for manipulating one train at a time, guiding it to a targeted destination on the track. However, even this limited functionality requires significant complication, as the global pathing server must:

- Perform path routing when a train wants to move to a particular destination.
- Keep track of the rough location of the train at all times.
- Keep track of what sensors a currently active train is expected to reach.
- Set switches and organize the train layout to line up with pre-configured routes.
- Calibrate the velocity, acceleration and stopping distances of trains on-the-fly as movements occur.
- Start and stop trains at precise times with precise commands to get them to perform precise movements and stop at predetermined positions.
- Decide when local pathing workers can be unblocked.

Very roughly speaking, the global pathing server looks like this:

```
void Planning::global_pathing_server() {
    Name::RegisterAs(GLOBAL_PATHING_SERVER_NAME);
    TrainStatus trains[NUM_TRAINS];
    Dijkstra dijkstra = Dijkstra(track);

    int from;
    PlanningServerReq req;
    while (true) {
        Message::Receive::Receive(&from, (char*)&req, sizeof(req));
        switch (req.header) {
        case RequestHeader::GLOBAL_CLEAR_TO_SEND:
        case RequestHeader::GLOBAL_COURIER_COMPLETE:
        case RequestHeader::GLOBAL_STOPPING_COMPLETE:
        case RequestHeader::GLOBAL_STOPPING_DISTANCE_START_PHASE_2:
        case RequestHeader::GLOBAL_LOCATE:
        case RequestHeader::GLOBAL_LOOP:
        case RequestHeader::GLOBAL_EXIT_LOOP:
        case RequestHeader::GLOBAL_SET_TRACK:
        case RequestHeader::GLOBAL_CALIBRATE_VELOCITY:
        case RequestHeader::GLOBAL_CALIBRATE_ACCELERATION:
        case RequestHeader::GLOBAL_CALIBRATE_STARTING:
        case RequestHeader::GLOBAL_CALIBRATE_STOPPING_DISTANCE:
        default: {
            Task::_KernelCrash("Invalid request: %d at global pathing\r\n", req.header);
        }
        } // switch
    }
}
```

As you can see, there are quite a few potential cases that the global server has to deal with, hehe...

4 User Guide

Our kernel and programs can be initialized as given in the README, the steps in which I will echo here:

1. First, download and build the main executable.
   (a) Use git to download the code from our repository:
   `git clone ist-git@git.uwaterloo.ca:t28cai/cs452-microkernel.git`.
   (b) `cd` into the source directory:
   `cd cs452-microkernel/src`
   (c) Run `make opt` to build the code into an optimized, loadable, executable image.
2. Next, load the code onto the Raspberry Pi, and boot up the necessary programs and wire connections.
   (a) Run `/u/cs452/public/tools/setupTFTP.sh <barcode> kernel8.img`
   to load the code onto the Raspberry Pi with barcode `<barcode>`. The barcode will be one of (currently) four, of the form CS01754[0-4], although only machines CS01754[01] are connected to a train set.
   (b) Before opening the command prompt on the control PC, run
   `logRPI.sh <barcode>`
   to connect the PC to the Raspberry Pi, if it is not already connected.
   (c) In another terminal on the control PC, run `gtkterm` to open a GTK terminal, which shows the command prompt of the A0 program as well as diagnostics and other important information. If there is already a GTKTerm open, this step can be skipped.
   (d) Finally, reset the Märklin box by holding the STOP and GO buttons for two seconds, then turn on the Raspberry Pi (or turn it off and then on again if it's already on).

3. At this point, the program should be showing a command prompt with a display, akin to the following:

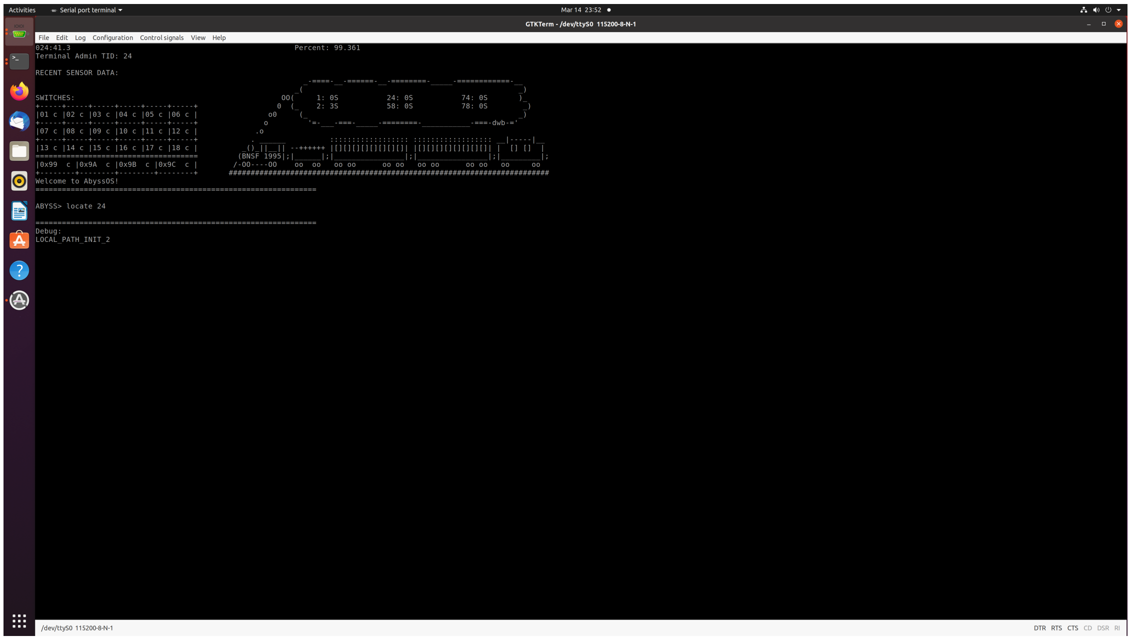

Figure 2: A sample image of our terminal output after TC1.

This image shows the GTK terminal command prompt taking up the whole screen on a computer connected to a Raspberry Pi. Pictured on the screen, roughly from the top left down, we have:

(a) A timer of the system time from power on, which includes minutes, seconds and tenths of seconds,

(b) The terminal admin task's TID, which we use as a sort of system diagnostic,

(c) A queue of recently activated sensors in red,

(d) A table showing the current status of the track switches on the track,

(e) An ASCII drawing of a train (courtesy of Donovan Bake and the ASCII Art Archive),

(f) The actual command prompt, where commands can be entered in next to `ABYSS>`, and

(g) A debug window, where debug output is sent to. When debug output exceeds a certain number of lines, it will scroll in the windows without affecting other parts of the screen.

When running the train on the track and using certain commands, the debug log will populate with status messages displaying the current actions and status of whatever train may or may not be running. These messages can include:

- Path printouts, which show the entire prospective route a train is preparing to travel on if a go-to-destination command is inputted,
- Current location status updates, including errors in calculations,
- Error messages if the train has encountered an error while running.

4. Once the prompt displays (the boot sequence may take a minute or two) commands can be entered. Here are the commands that are accepted by the terminal:

   - `tr <train_number> <train_speed>`

     Sets the train with label `<train number>` to the desired speed (0 is stop, 1-14 are speeds of increasing degree, 15 reverses the train). 16 can be added to any of these commands to turn on the lights of the corresponding train.

   - `rv <train_number>`

     Reverses the train with the given train number. This command may take a few seconds to process, as the train needs to fully slow down before reversing.

   - `sw <switch_number> <switch_direction>`

     Changes the switch with the given switch number to straight (s|S) or curved (c|C). This command also issues a command to shut off the solenoid to the switch after 200ms.

   - `go <train_number> <destination node> <offset>`

     Makes the train with the given train number move to the given destination node (input as a number from 0 to 143, inclusive) assuming that the path is possible (an error message is printed otherwise). The train will attempt to stop as close to directly on top of the node as possible, plus or minus the offset (in mm). Offset is optional (default 0).

   - `locate <train_number>`

     Moves the train with the given train number forward slowly until it hits a sensor, in an attempt to assign a definite location to the given train.

   - `loop <train_number> <speed>`

     Makes the train with the given train number enter a large loop around the track, and loop around at a specified speed. If speed is not given, the speed will be set to the maximum speed level that we support.

   - `exloop <train_number> <destination node> <offset>`

     Makes the train with the given train number exit looping and attempt to move to the given destination node (input as a number from 0 to 143, inclusive) if the path is possible (messages are printed otherwise). The train will attempt to stop as close to directly on top of the node as possible, plus or minus the offset (in mm). Offset is optional (default 0).

   - `(cali|base|accele|sdist) <train_number>`

     Calibrate various attributes of the train, including velocity, acceleration and stopping distance.

   - `init <train_number> <args...>`

     Does nothing but print a debug message right now. But hey, it's useful for testing the UI :D

   - `q`

     Quits the kernel and jumps to a restart instruction in assembly.

   Any train number higher than 99 or switch number higher than 156 will be rejected, although only train numbers 1, 2, 24 and 58 and switch numbers 1-18, 153-156 actually exist and function.

   Commands for train numbers that do not exist will be rejected. Commands for switch numbers that do not exist may or may not be rejected, as the error checking only checks that the number is in the range 1-999, inclusive.

5 Calibration Experimentation (TC1)

We initially considered doing and using manual measurements for important train constants, hardcoding them directly into the kernel and depending on them for localization and decision making, but quickly decided against it in favour of semi-automatic calibration for a number of reasons:

- The status of the track can change constantly, which means that some form of dynamic calibration is essentially mandatory.
- Automatic measurements are likely to be much more precise than human-based systems.
- Measuring things by hand sucks. This way, you can leave a train at the track for 15ish minutes while you go grab a coffee, and the measurements will be ready when you return. Maybe.

The result of this is that our kernel supports commands for extensive train calibration systems, of which there are 4.

5.1 Base Velocity Calibration

Before we can do any other calibrations, we first need to calibrate the values for a "base" speed, i.e. some reliable, slow speed that we can bootstrap other calculations off of. Ideally, this base value should be a value low enough that velocity calculations done using the speed can treat acceleration as negligible, which greatly simplifies calculations. For our measurements, this speed is speed 3 for train 24, and speed 4 for train 78.

Given the choice of the base velocity calibration speed, upon receiving the instruction to perform a base velocity calibration, our system first navigates the train to a fixed sensor, then slowly moves it up until it hits another sensor, then sends the train in a long loop at the base speed and uses sensor readings and the internal clock to calculate the base velocity. This base velocity is then reported to the terminal.

5.2 Regular Velocity Calibration

After the baseline velocity is known, we can begin calculating the speeds we would like our trains to actually run at. For simplicity, each train only supports three different speeds: stopped, a low speed, and a high speed. To test these speeds, we simply run the train in a large loop at the desired speed and use sensor timings and known distance differences to calculate velocities.

Importantly, because speeds seem to vary slightly depending on what the PREVIOUS speed value was, our middle speed value has to support two different values: one if the previous speed was higher, and one if it was lower. Both of these possibilities are calibrated for in our calibration procedure.

5.3 Acceleration Calibration

Acceleration calibration is very complicated. Each acceleration calibration operation consists of taking two fixed speeds, changing between them, and carefully using sensor timings, known distances, known velocities, then solving sets of kinematic equations to come up with the acceleration values:

$$t_1 = \frac{d - v_2 t}{v_a - v_2} \qquad\qquad a = \frac{v_2 - v_1}{t_1}$$

where d is the distance measured, v_1 and v_2 are the known initial and final velocities, v_a is the average velocity, and t is the rough time taken. In essence, we linearly interpolate acceleration between velocity values and use complicated equations to solve the problem of having very limited information.

On the trains side of things, a loop is used to navigate between values, as before.

Also, decelerating to a stop is not tested here, as those are measured during stopping distance calibration.

5.4 Stopping Distance Calibration

Measuring stop distances is similar to measuring acceleration, but because we are not guaranteed to hit a sensor after sending the stop command, we must use the base velocity to get a sense of our overall position.

When the global pathing server receives a command to begin stop distance calibration, it first sends a train into a long path, long enough that the train has sufficient time to accelerate to a steady, fixed velocity. It then stops the train, and moves through a series of stopping stages to obtain more useful measurements, from which it can perform more calculations:

$$d_{\text{stop}} = d_t - d_b \qquad t = \frac{d_{\text{stop}}}{v_a} = \frac{2d_{\text{stop}}}{v} \qquad a = -\frac{v}{t}$$

where:

- d_t is the total traveled distance,
- d_b is the "base distance"; the distance traveled at the base velocity after stopping, which is used as a sort of ruler,
- v is the fixed velocity from which we want to measure stopping,
- $v_a = v/2$ is the average of the fixed velocity and the stopping velocity (0),
- d_{stop} is the stopping distance,
- t is the stopping time (assuming linear deceleration), and
- a is the deceleration value (also assuming linear).

5.5 Recorded Values

Our experimentally recorded values for trains 24 and 78 are stored in `tc1/calibration.csv`. We do not have values for other trains, and as such cannot use them in demonstrations, but we could with about 20 minutes per train of calibration time.

The units of the measured values are as follows:

- All distances are in hundreds of millimetres.
- All velocities are in hundreds of millimetres per second.
- All acceleration values are in hundreds of millimetres per second squared.

This particular choice of unit was chosen because it strikes a reasonable balance of precision, prevention of loss of accuracy under the constraints of integer arithmetic and not being so big that they're impossible to interpret.

6 Acknowledgements

Special thanks to:

- Mike Krinkin for his excellent ARMv8 tutorial, available here. Without it, writing a stable context switch would likely have been much more difficult.
- John Wellbelove and other contributors for their work on the Embedded Template Library (ETL) which we have shamelessly lifted directly for use in our kernel thanks to the wealth of data structures it provides.
- Marco Paland for his work on reimplementing `printf`, a function that is incredibly useful for debugging purposes, and requiring only a simple `putchar()` implementation to function properly.
- Rock Zhang (@codingbelief on GitHub) for his comprehensive ARMv8 tutorial, which was invaluable for setting up the MMU.
- Darwin Chen for being the brave soul who first plundered into the depths of MMU configuration territory, and who provided hope in dark times when things weren't working properly.
- Donovan Bake and the ASCII art archive for their lovely train art.

7 Appendix A: The Memory Management Unit

Though the MMU itself is not spectacular, it is the required hardware configuration if we want to enable some form of data caching. Thus, the minimum requirement is a translation table that flat maps address directly to each entry on the table.

we will not talk about many details related to MMU itself, but more about what design decision that we thought works the best. If I go into detail why certain paramter are set a certain way, this will take about 10 pages to complete.

1. We used a level 1 table and four level 2 table for flat-mapping. Once again, if the goal is to maximize performance through stuff like TLB we might bave to go in deeper, but for our purpose this is enough
2. since the only interesting memory address are the first 4 gb (first 2gb is real, while device memory is located somewhere in the 4th gb). the goal is to map them according to their usage
   (a) Our code is located at 0x80000, the default start location of pi-4, and for convenience reason, we will map the entire first 2mb block from 0x0 to 0x100000 as execution memory, giving el1 the permission to write/read/execute and el0 the permission to only execute. Linker will ensure this area only contains code which we execute. note that our current executable is only about 46 kb when it comes to .text section, thus this area should be more than enough to contain all the code
   (b) Future byte all the way until 2gb mark is simply marked as memory that can be read/write for both exception level, in the linker I assured any .data or related content would be moved ot this area. This include global constants and user stacks. We also choose both inner and outer write back Non-transient, this is because we seems to easily max out the L1 cache on processor 0, and most of our operation could be contained within the 2mb storage provided by L2 cache. this means very little use of actual memory is used, so we don't have to worry about writing back to memory (also there is no multi core so there is no cache coherency).
   (c) Some part of the memory are device specific memory. They are not real memory addresses, but the physical location will magically convert processor access into the right register. Thus, this area is similar to marking as read/write for both exception level, but we need to avoid caching. If a write to register is cached, it essentially have 0 effect.
3. it also worth noting that we intentionally turned off TCBR1_EL1 which is suppose to point toward the upper address beyond the 4GB, but since we will never access that region in correct execution sequence, it nice to turn it off. in case of a bug, we will at least get some format of exception.
4. we didn't have a level 0 table because there is no need to support one, the level 1 table already support up to 512 gb of memory mapping, and I am only using 4gb. (unfortunately level 2 table only support 1gb so I couldn't go further, however, there are some setting you can tune such that table will only use 4 entries)
5. funny enough, the 3rd table (for the 3rd gb) is completely irrelevant to our stuff, there is no real memory there and there is no device memory that relies on it, so we are free to actually just, not even initialize it and leave it blank, but I kinda just filled it in anyway

that is pretty much most of the design, later on we simply setup the related parameter to true and flip then switch on STCLR_El1 and everything would start running.

Lastly, we stashed the table starting from 0x40000, and point TCBR0_EL1 toward it then we have a properly working MMU, though just flat mapping. With MMU, we have dcache, with dcache, our SRR time went from 31 micro seconds to almost < 1 microseconds.