UNIVERSITY OF WATERLOO

MicroKernel Architecture vTC.01

Formal Process Report

Date: Tuesday 12th May, 2026

by

Hans Bhatia (*user_id* : *h23bhati*)
Conor Lamont (*user_id* : *clamont*)

repository_link: h23bhati1/cs452-kernel
current_commit: 46b4bbf7

University of Waterloo

Contents

1 Kernel Structure

(A) The kernel operates on the kernel stack, it uses the single stack approach.

(B) Allocates stacks for users on global memory.

(C) After hitting the exception vector, we transition into a C++ exception handler that does some basic error handling (differentiates between device interrupts and synchronous interrupts, etc).

(D) Uses round robin scheduling with multiple priority queues and scans down the list from highest to lowest priority when choosing the next task – does involve starvation...

2 Program Structure

The header files (.h) are self-explanatory as they are a subset of the implementation files (.cpp). We will omit the file extension for C++ files (they can be either .h files or .cpp or both). It has been setup to allow for nesting up to 2 directories. This mostly omits testing related files. The structure of the repo is as follows:

2.1 Assembly files (aarch64)

`Boot.S`

- The entrypoint of execution. Initially, we transition to kernel mode, use the `vbar` instruction to set the start address of the exception vector table, and boot into the kernel.

`Irq_vec.S`

- Contains the exception vector table, which is aligned. Subsequently, it contains the switchframe logic for a user task and stores the exception segment it propagated from.
- Contains the logic to deal with the case device interrupts.

`Switchframe.S`

- Consists of the switchframe logic from kernel to user mode.
- Consists of the switchframe logic to reload back into a task that was interrupted.
- Contains the latter half of the switchframe logic from user to kernel mode.

`Cache.S`

- Consists of cache cleaning procedures.
- Exposes routines to switch between cache modes (icache, dcache, bcache, nocache)

2.2 Kernel related tasks (k-programs)

`init-task`

- Contains the init task, the first task run after bootstrapping the kernel.

`task-wrapper`

- Contains the task-wrapper which wraps all the created user tasks and makes sure to call the `Exit()` syscall at the end of function execution.

`clock-task`

- Contains the clock-server, a task that is never blocked if work is available.
- Contains the clock-notifier, that constantly awaits the fire of a clock tick event.

`idle-task`

- The idle task... As of now, this keeps the kernel alive.

`turnout-task, train-task`

- Tasks to perform the work of setting turnouts and sending commands to trains.

2.3 Library (lib)

`clock-server-interface`

- Wrappers around srr for interacting with the clock server (Time, Delay, DelayUntil) and defines constants used by the server + client.

`event-lib`

- Lists all possible events that can be awaited upon. (acts as a shared lib for synchronizing event ids)

`rps-helper`

- Helpers for running the rps tests and defines constants used by the server + client.

2.4 Data structures (data-structure)

Contains data structures(freelist, linked-list, message, queue, scheduler, task), explained in data structure section.

2.5 System files (system)

`clock(time, sleep)`

- Exposes a sleep routing used only for testing (warm reload for benchmarking).
- Provides functionality to get the current rpi free counter value in different forms of precision (milliseconds, microseconds).

`cache`

- Provides a C interface to set desired cache modes (icache, dcache, bcache, nocache).

`Hardware interfaces (rpi, console, gic)`

- Exposes printing logic and I/O config logic.
- Exposes a gic interface that initializes the gic for desired device interrupts, checking the type of generated interrupt, and cleaning up an interrupt.

`Syscall interface(syscall)`

- Describes the svc codes to use and implements the interface for a user task to call them.

`Output interfaces(console, debug, marklin)`

- Helper functions to do specific types of i/o. We also added a Debug flag to enable/disable debug output.

2.6 Kernel files (kern)

`Bootstrap (kernel_bootstrap.cpp)`

- Contains the main loop.
- Includes bootstrapping logic – initialize the data structures for task descriptors and stacks.
- Setup task queues, device interrupts, and event handlers.

`Syscall handler (syscall)`

- Handles an incoming synchronous interrupt request.

`Tasks (task)`

- Handles calling the init task, creation, deletion, activation, getters/setters for task descriptors, keeping track of the curr task.

`Device Interrupts (device-interrupt)`

- Handles interrupts generated by the gic (asynchronous exceptions). Identifies, cleans, and dispatches the interrupt to the necessary event handler.

`Event Handler (event-handle)`

- Handles generic events. Facilitates the internals for the `AwaitEvent` system call.

2.7 Utilities (utils)

`Math, String`

- Routines for math and string operations.

`Message`

- Provides a typed message wrapper for the SRR routines. Currently supports Int and String types.

`Kassert`

- Provides a basic way to assert certain conditions for testing.

`Global constants (constants)`

- Exposes redundant constants for the repository.

2.8 Servers (p-servers)

`name-server`

- Contains the code related to name-server setup + runtime code.

`rps-server`

- Contains the code related to rps-server setup + runtime code.

`uart-proprietor-server`

- The backend server and proprietor for uart communication. Templated for both Marklin and Console.

`marklin-controller`

- A collection of files used for controlling the marklin. For example, we have our train master, our master task for tracking the state of trains and switches on the track.

`train-control`

- A folder containing files for train control including sensor attribution, collision avoidance, routing and track reservation.

2.9 User Tasks (p-programs)

`performance`

- The task that sets up performance testing of SRR routines under the 12 conditions described in the K2 assignment specification.

`sample-user-task`

- The task used for the K1 assignment routine.

`Generic Server Clients (rps-client, clock-client)`

- Generic tasks that interact with the servers as required by the assignment specifications.

`train-task`

- Templated tasks to run commands for specific trains, track their state, etc.

`train-io-task`

- Tasks to deal with input to the Console and then direct it to the appropriate place to be sent to Marklin, or input from Marklin.

`timer`

- task which outputs the current time the program has run

The rest of the programs are used for testing.

3 Kernel Flow

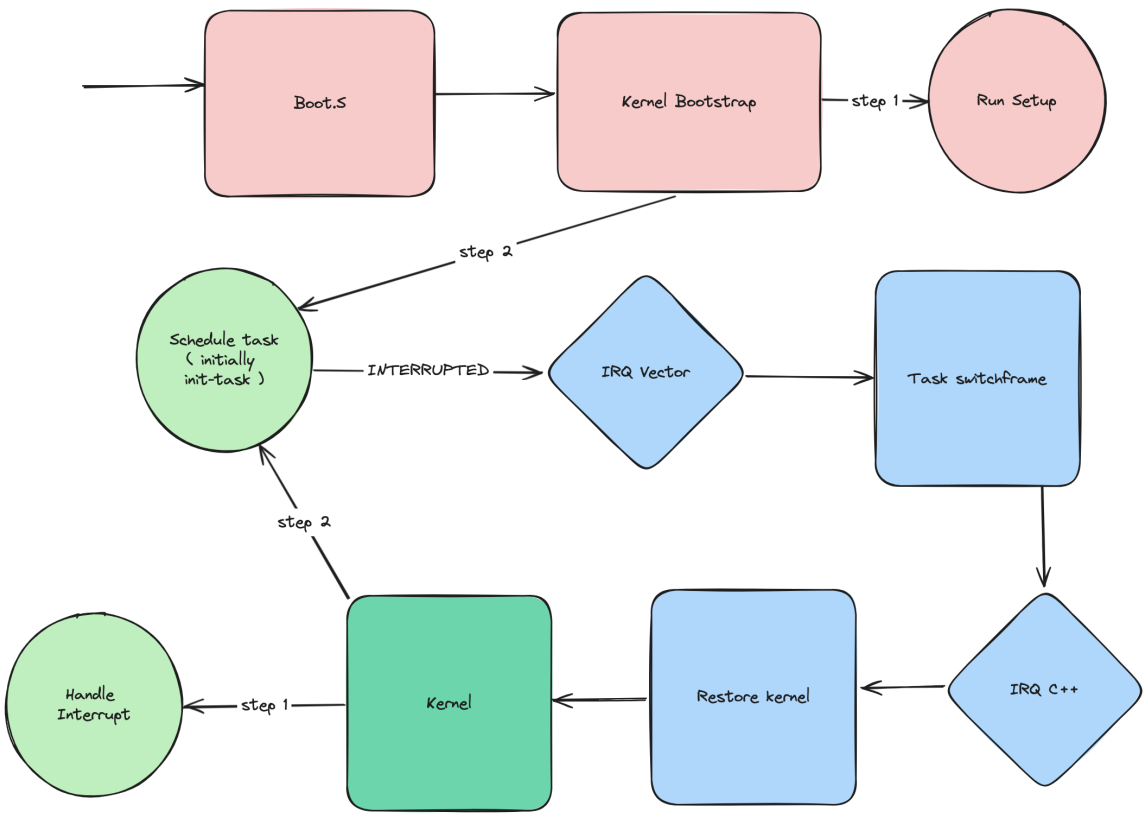

Figure 1: Kernel Flow - simplified global flow of the kernel

4 Data Structures

4.1 Task Descriptors

The task descriptor is relatively simple – it has all the fields one would expect like the task id (tid), parent tid, priority level (with levels ranging from Low to Kernel, see `task-priority.h`), a bool indicating readiness, a pointer to the top of its user stack, a return value field, a pointer to its args and its function pointer. The ready state field is still not in use; however, we kept it in the class for possible future use. See `task-class.h`.

4.2 Freelist

To store all the task descriptors, we made a free list of length 100. We chose this number as this should be much more than a small train-system kernel will ever need. The free list is simple – it is essentially a large linked-list (stored as a fixed length array) which has a pointer to find the next free slot in $O(1)$ time, and each slot is linked to the next free slot in the list. Freeing is also very simple as we simply mark it as free and add it to the list (in our implementation, the front). The actual structure itself is implemented in `freelist.h` as a template, and each freelist item is an instance of the `freelist_item<T>` class defined in `freelist-item.h`. The freelist-item class simply contains an instance of an object of type `T` and the index of the next free object. Of course, in our case, the generic `<T>` type is the task class. We implemented this with generics in case we ever want to use a freelist elsewhere in our kernel.

4.3 Queue

We implemented a standard FIFO queue (again using generics) in `queue.h`. This was used for our task scheduler at each priority level – 1 queue per priority, see below. Each queue is of length 30, respectively, as this is well more than we should ever need among all tasks in our kernel, let alone a single priority level.

4.4 Scheduler

As mentioned above, our scheduler contains one FIFO queue for each priority level. As specified in the kernel description, the highest priority task available is always selected for the next task, meaning higher priority tasks starve lower priority ones until they exit. Since we use FIFO queues, the task that has been waiting the longest is selected at the highest level available. After no task is available at any priority level, our kernel waits for input and returns.

4.5 Message

The message class is a template class for tasks to do message passing which consists of 3 fields

1. char * data: The actual data to be sent in this message.
2. int data_len: The length of data.
3. T sender_tid: This is template type T because we specify this as either an int* or just an int depending on whether it's a Send, Receive or Reply.

4.6 Linked List

We wanted to include a sorted linked list for efficiency when it came to handling multiple clients within servers such as the clock server. We implemented this with a C++ template with arrays of fixed size. These seemed to be a simple way to reduce the time taken in the clock server which would in turn reduce waiting time of the clock notifier - and reduce the possibility of missing any ticks.

5 Unimplemented Aspects

Train routing and stopping generally work, but the calibration is limited. We intend to expand our calibration to multiple trains on both tracks.
There are some bugs still under investigation by our dedicated QA team. There are a lot of TC2 bugs and unimplemented aspects. See TC2 Quirks.

6 Known Quirks + Assumptions made

(A) Some output to the console does not complete on the kernel-side if the kernel exits as soon as all the tasks are done. We decided to exit only after user input of any key at the end.

(B) We assumed that a syscall will not resume the same task, instead add it to the back of its respective priority queue, and proceed with the next task.

(C) UART handling: currently, per line, we have 1 uart proprietor, 1 uart backend server and 2 notifiers. Each notifier awaits a single event and notifies a single client.

(D) TC2 Quirks: There are a lot. A lot of the spec does not work on the Marklin track. Our stopping is also worse than our tc1 stopping, often failing. Reverse routing sometimes works.

7 Parameter Choices

(A) Stack size (8 mb) – similar to linux default sized stacks, atleast according to this stack exchange post. post

(B) Queue length (30): each priority level has a queue of length 30 (holds 30 tasks per priority level). This is much more than we should need to run our train system.

(C) Freelist length (100): The freelist length (number of task descriptors) is 100, which is well more than the number of tasks we should need, even among tasks of all priorities in our system.

(D) Number of k2-performance test runs (200): For any given test setup (optimization, caching, send or receive-first, message size) we run 200 test runs as this gives enough samples to get convergence, negate outliers and also negate the time it takes to make the clock calls at the beginning and end.

(E) Inbox length (10): For message passing, each task has an inbox of length 10. This seemed adequate because we do not have many tasks, and any given task can only send one message at a time before said message is received and then replied to (as Send is blocking).

(F) Event Handling (1): For event handling we assume there is only one task awaiting per-event. Using a notifier-centric approach we can allow for multiple clients to wait on events (however, this does not seem necessary at present).

(G) Linked-list (length 10): We chose our linked-list of blocked tasks in the clock server to be of length 10 (see clock-server.cpp) as we only have 4 idle tasks for now and it seems unlikely we will ever have more than 10 tasks blocked concurrently.

(H) Tick delay (10 ms): We used a 10ms tick for our clock interrupts as this is a nice, base-10 number and although 1ms also works, it makes debugging the kernel more difficult as logging from the kernel is quite slow and the 1ms tick is time-constraining.

8 Performance Measurement

8.1 Process description

For our performance measurement, we first sleep 20s (busy-wait) to allow the CPU to warm up to a constant temperature. We then used two simple programs for each test, simple_send() and simple_receive_and_reply(), which respectively send N times and receive/reply N times. We chose N=200 as this seemed adequate to get a valid number of trials to achieve an accurate average, and it is also large enough to make the calls to clock() negligible. We used assembly functions to enable/disable the caches (and clean/invalidate as necessary) so that all the tests could be run at once for a given optimization level - we also ran tests where we rebooted and then changed the cache setup statically at boot and achieved approximately the same results. See k2-performance.cpp.

8.2 Results

We will discuss each of the 5 variables that were measured and the apparent effects they had on our performance results.

(A) **Optimization**
Optimization had major performance benefits for our program. For instance, Send-First with both caches enabled for 64-byte messages took our program 72 microseconds when optimized but 514 microseconds without optimization! Other setups had similar differences.

(B) **Cache Type**
Caching had a notable impact on our performance results, but not to the same extent as optimization. Looking at optimized, send-first, 256-byte message results, we see that the test with both caches enabled took 131 microseconds, both disabled took 203, data cache enabled took us 203, and instruction cache enabled took 132. Other setups gave similar results.

(C) **Send or Receive-First**
Send first vs receive first had very little impact on our results. Generally, however, receive–first faster by a constant, negligible number of microseconds (in the range of 2-3).

(D) **Message Size**
As one might expect, message size had quite a notable impact on performance. For instance, optimized tests with both caches enabled in Receive-First took 51, 69, and 131 microseconds respectively for 4, 64, and 256-byte message sizes. The same tests with non-optimized code took 235, 482, and 1237 microseconds, respectively.

8.3 Conclusions

Optimization is the variable that affects the performance of our system the most. This makes sense as we are comparing no optimization to O3 (highly optimized). Message size has a similar level of impact on our test results, which also makes sense since larger message sizes mean the kernel has to copy larger amounts of data for each message. Our results indicate that the instruction cache being enabled seemed to have quite a positive impact on performance, while the data cache being enabled had very little impact. Send or Receive-First has a constant, very small impact on our performance such that send–first is a few microseconds slower for any test run - this is because our measurements start right before the first send on each run, meaning in send-first, the beginning of the receive function is counted in the time produced, but in receive-first, that code will have already run before the clock starts, giving a constant startup time included in send-first but not in receive–first.

9 UART

9.1 Server Setup

We have 1 backend server and 1 proprietor per line. The Console has an RX notifier and a TX notifier, Marklin has RX and CTS notifiers. We use a state machine to ensure the Marklin is truly ready before sending to it. Interrupts are disabled via the IMSC register when they arrive, and they are enabled once an awaiter has begun to await for said interrupt.

10 Train Control (TC1 + TC2)

10.1 Task Setup

TC2 - We have multiple modular tasks to deal with train control. We begin with a sensor tasks that requests a sensor-attribution-server for the stakeholder for a new sensor update. With these stakeholders, if a train is already routing(has a destination) we update the path if possible by notifying a router server with the sensor and the affected train. Next, the router server manages tracks and reservations - where any train commands are forwarded to a respective train server (one per train).

TC1 - There is 1 server that tracks the total state of the track including train locations, speeds and turnout values - the train master server. There are worker tasks which send to the train master, such as train tasks, a polling task, the UI command line handler, and a task to control turnouts. The master server is running at a relatively low priority, and all the workers are above it (train tasks, turnout tasks, etc. except for UI command handling). The polling task runs at quite a high priority, above other workers, to ensure we get polling data quickly. We poll about once every 140ms (60ms poll, 80ms delay).

route the train, and how you get the train to stop at a target location. It should describe how you are modeling the train, including a description of any calibration data you have collected for modeling, and a description of how it was collected. Finally, it should describe how your train control system is implemented. In particular, it should describe the purpose and design of any tasks that are running, and their priorities. If you have made any modifications to your kernel to support TC2, explain

10.2 Tracking

TC2 - For the initial tracking we fix a starting sensor, namely A13, at A13 we stop and wait for any commands (such as free-start, or go to a path). We track trains by estimating the distance from a train to its next sensor, we use our estimated velocities of the train to estimate the distance travelled from the last time travelled. This in conjunction with reservations will guarantee that the train will be the closest.

TC1 - We track trains by storing the most recent sensor read for any train. Any time we hit a sensor we iterate through the trains to determine who was supposed to hit that sensor next(if any) and update that train if so. A train is first activated, i.e. we associate that train with the next unexpected sensor hit, with the command "fd [train#]" for "find".

10.3 Routing

We use dijkstra's algorithm to find the most optimal path to a given node on the track. We look ahead by 2 sensors from the previously read sensor of a given train so that we do not route to paths that are too close to stop at. For **reverse** routing we simply augment the dijkstra implementation to assess the reverse node and we give it a slight weight of 20 (somewhat of a magic number). We also compute an action event queue which will pop an event on certain sensor hit.

10.4 Stopping

TC2 - we decided to go with more of a reliable approach due to the low amounts of track time we had to sample + model data. Instead, a sensor before, we decelerate to a low speed, we then abruptly stop when we receive the sensor hit for the destination.
WE GOT RID OF THIS IN TC2 – We stop in the following manner. When we first route, we store the distance of the whole path. Every time we hit a sensor we subtract the distance we travelled since the previous sensor. When we are close enough (i.e. we hit a sensor where the next sensor will be too close to the destination to stop on-time), we will stop with a time delay from the current sensor. This is calculated such that we will travel to exactly our stopping distance from the destination sensor in that period, given the velocities we seed into our model.

10.5 Priorities

In terms of priorities we just estimated the frequency of use of our tasks. We also make sure to avoid any srr deadlocks.

10.6 Data Sampling

See the **p-programs/train-speed-sampler** folder in our repos for the code used. We commented the code as it became deprecated after changes to our interfaces.

However, we found our acceleration measurements were not great, so we then decided to manually measure stopping distance at given speeds by immediately sending speed 0 commands after a given sensor hit and measuring the physical distance travelled by the train.

The velocity and stopping distance measurements helped us seed our models, stored in p-servers/marklin-controller/track-constants.h.

Please see the averages for velocities collected. WE also have raw data that will be committed but is not yet in the repos (will be on master in a commit, soon).See raw-data folder.

11 Implementation Comments (K4)

1. We have a busy-waiting debug namespace which is disabled for submissions. This is done via a DEBUG flag in our Makefile, which makes our debug functions do nothing (using #ifndef). After this, the only busy-wait logs we use are to initially set up our terminal prior to the spawning of the i/o

servers. Then, every i/o operation is non-blocking.

2. In order to combat busy wait we spawned I/O servers that rely on notifiers to let them know when they can send/receive any input. This avoids the occurrence of polling in the server.

3. Note: the task takes some warm up time to setup the track switches and the display the user interface. We request that this stage be left alone.

12 Program Output

12.1 The Clock Server/Client test

See figure 4.
A single type of client task tests your kernel and clock server. It is created by the first user task, and immediately sends to its parent, the first user task, requesting a delay interval, t in ticks, and a number, n, of delays. It then uses WhoIs() to discover the tid of the clock server. It then delays n times, each time for the delay interval, t. After each delay it prints its tid, its delay interval, and the number of delays currently completed on the terminal connected to the ARM box.

We run four concurrent instances of these, with the delay intervals and number of delays specified in the kernel 3 description.

Firstly, they will output their tid and state their delay interval and number of delays. E.g. "**Clock client with tid 5, delay interval 10 and 20 delays started**". Then, each clock client will respectively print after each of their delays e.g. "**Clock client with tid 5 waited for 10 ticks. Completed 2 of 20 delays**". We also print (in yellow, highlighted text) "**=== 100 ticks past ===**" which is printed by the clock server every time 100 ticks pass. This can help separate output so that it is readable and also shows progression in the system.

12.2 RPS game test

See Figure in the Gallery for example output. Our RPS server initializes at tid 3 which is output in the first line. Then, the clients find the server via WhoIs() calls - see the lines at the top saying **"RPS client n located RPS_SERVER at tid 3"**. Then the clients sign up, and they are all successful, see the outputs **"RPS client tid n successfully signed up"**. The one exception to this is tid 5, which is a special case; tid 5 successfully signs up then tries to sign up for a second time, and the server realizes tid 5 is already in a game, see the output line **"RPS Server received Play request from tid 5, but they are already in a game"** (about 10 lines above the first yellow line). Then the first yellow text is output by the client tid 5, **"Task 5 trying to signup second a second time... not allowed ..."** which shows tid 5 was rejected to sign up after it has already signed up. The server also outputs when it starts a game between two tids, see the lines near the top **"Getting a game started between P1(tid: x) and P2(tid y)"**. Clients will output whether they won or not. When you see "Task 4 won!" for example, you should look for who task 4 was playing via the aforementioned "Getting a game started.." logs (in this case, tid 4 played tid 5). To see what moves lead to "Task 4 won!", you should look at the 2 most recent server outputs for the two tids in that game, in this case **"RPS Server received msg Paper from tid 4"** and **"Server received msg Rock from tid 5"**, which means tid 4 played rock and tid 5 played scissors, which lead to tid 4 winning. For ease of reading, the client will also output **"Client n received result (Win/Draw/Loss) after playing move (Rock/Paper/Scissors)"**. Note that output will vary because moves are randomly generated based on clock values.

12.3 Idle time reporting

See Figure 6.
We perform idle time reporting in the kernel. We keep track of the total time the idle task has run by calculating the time before and after an idle task is started from the kernel (before and after the context

switch) and adding this onto a running counter. We then calculate (total_idle_time / total_run_time) where total_run_time is the time since the idle task was first run - therefore our percentage will start around 100 since after the first run, the overall start time and the current run's start time will be about the same. You will also notice it drops a bit to about 90 while other tasks are running (the clock client tasks), then once only the server/everlasting tasks like the clock notifier and clock server are left, our idle time reaches up to 98% total time, as shown in the figure. We only print this value when it changes from a previous log, so if it is not printing this means we are still calculating it, but not outputting it as to not clog up the terminal. You will see logs like "**Idle for 96%**".

12.4 UPDATE! K4 friendly UI

We now have a friendly UI divided up into boxes for commands, time, and track state (then console and debug logs at the bottom).

13 Additional Notes

(A) The syscall interface provides two additional syscalls namely `YieldNoReturn` and `CacheSelect`.

(B) `YieldNoReturn` is purely for development purposes and are **not** used in any of the assignment routines.

(C) `CacheSelect` is used for performance measurement as the cache can only be toggled in kernel mode - and we wanted the tests to be continuous for ease of recording data.

(D) Output to the console is blocking.

14 Gallery

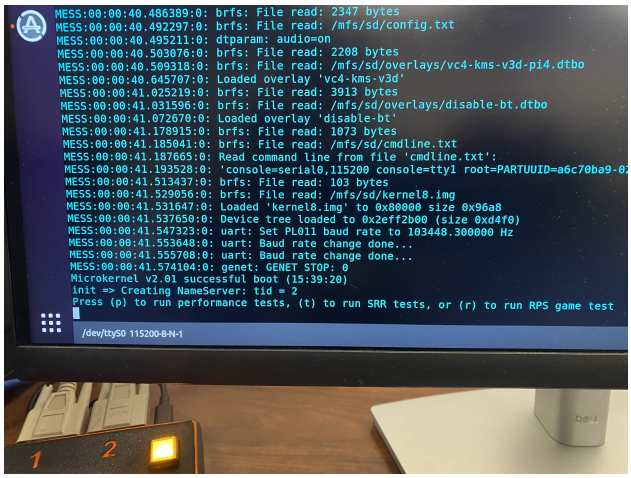

Figure 2: Options screen

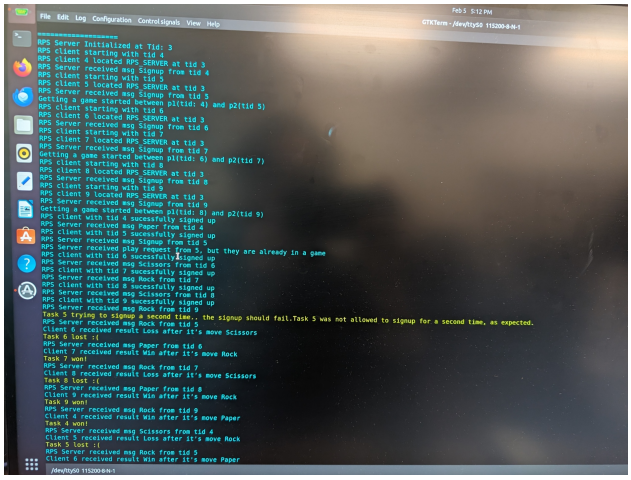

Figure 3: Rps test logs

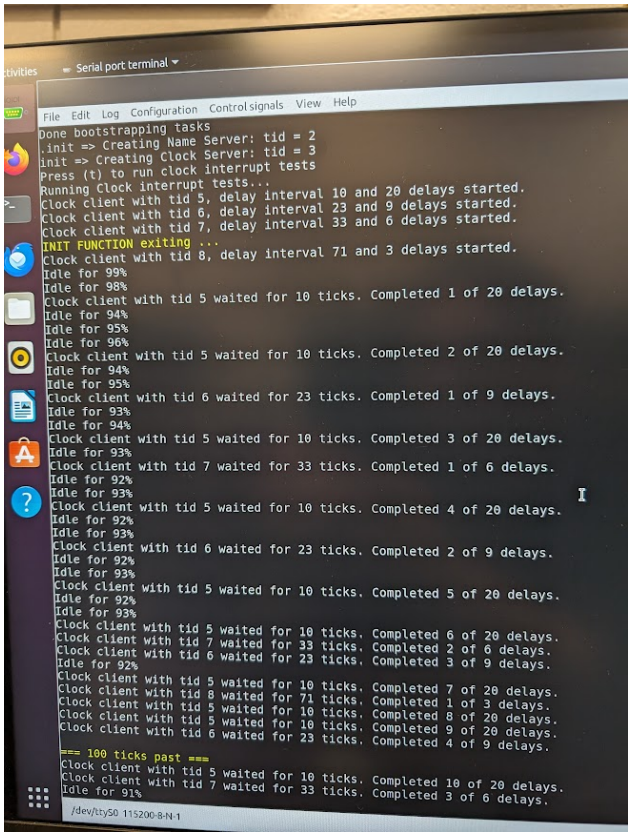

Figure 4: Clock server/client logs

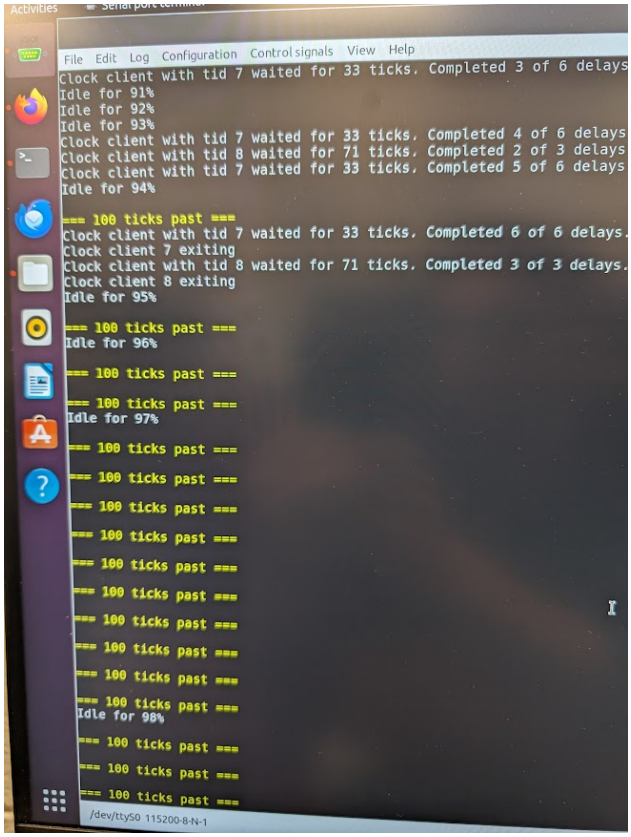

Figure 5: Idle task time reporting

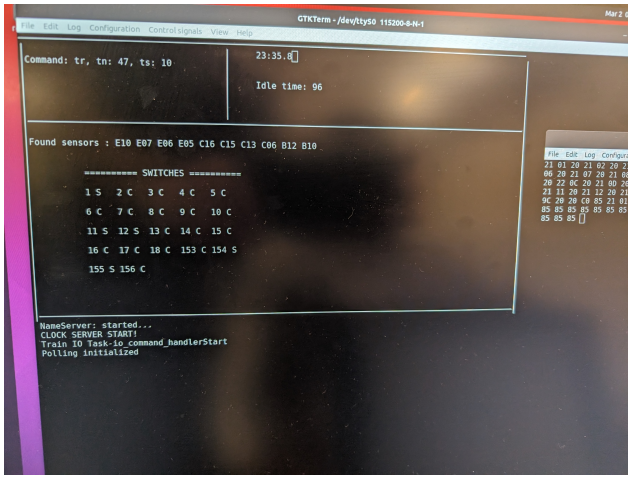

Figure 6: K4 (Isn't she lovely)