Kernel Assignment 4

Niclas Heun
nheun@uwaterloo.ca
21111245

Trevor J. Yao
t27yao@uwaterloo.ca
20830290

Professor Ken Salem
CS 452/652: Real-Time Programming (Fall 2023)
University of Waterloo
26 October 2023

Contents

1 Overview **3**

1.1 Submission SHAs . . . 3

2 Execution **3**

2.1 K2 Test Programs . . . 3

3 Kernel Structure **4**

3.1 Generic Kernel Loop . . . 4

3.2 Critical System Parameters and Limitations . . . 4

3.3 Algorithms . . . 5

3.3.1 Context Switching . . . 5

3.3.2 Stack Allocation . . . 6

3.3.3 Task Allocation . . . 6

3.3.4 Task IDs . . . 6

3.3.5 Scheduling . . . 7

3.3.6 System Calls . . . 7

3.3.7 Message Passing . . . 7

3.3.8 Interrupts . . . 8

3.3.9 Idle Task . . . 9

3.3.10 Clock Server/Notifier . . . 9

3.4 Data Structures . . . 9

3.4.1 Task Queue . . . 9

3.4.2 Task Descriptor . . . 10

3.4.3 Messages . . . 10

3.4.4 Name Server . . . 10

3.4.5 Event Queue . . . 11

3.4.6 Clock Queue . . . 11

3.4.7 Game Server . . . 11

3.4.8 Deque . . . 12

3.5 I/O Servers . . . 12

3.5.1 Märklin I/O Server . . . 13

3.5.2 Console I/O Server 14

4 User Tasks **15**

4.1 Kernel Assignment 1 15

4.2 K2 - Performance Measurement 15

4.3 Kernel Assignment 3 17

4.4 Kernel Assignment 4 18

4.4.1 Overview 18

4.4.2 Structure 19

1 Overview

This assignment begins the implementation of the Train Control Kernel in order to enable a task-based parallelism model for task management. The end goal is to enable more complex train-control, which would not be possible with a single sequential thread of execution, as used in A0.

For this assignment, there is no interactive interface. The Kernel starts up and executes the user-level entrypoint function `void user_main(void)`. The User level programs required for each assignment are found in `user/includes/kX.h`, where `X` is the assignment number, and the entrypoint executes the specified user programs.

1.1 Submission SHAs

The repository for the kernel is ist-git@git.uwaterloo.ca:t27yao/cs452-kernel.git. The submission SHAs for their respective assignments are listed below:

- K1: 6a8841836fc9ce629b573e94b37c901b4d71f2ee;
- K2: f284ab2f8195101c9d9bdd1c908a8abfd07324c3;
- K3: d638dfb64659a50951f2fbd32e720dfaa6aaa002;
- K4: 2702435fcfd212e159a1c71a05c4fdd220f6ed48.

2 Execution

The Kernel without any assignment specific user-level programs can be built from the root directory by running `make` or `make all`. This is usually not what should be done. In order to build the user-level programs for each Kernel Assignment, execute the target `make kX`, where `X` is the assignment number. It may be necessary to set the `XDIR` variable to point to the directory containing the cross-compiler. For example, in the `linux.student.cs` environment, `XDIR` should be `/u/cs452/public/xdev` (i.e. the parent of the `bin` directory). Therefore, the program can be compiled in the `linux.student.cs` environment by changing directory to the program root and executing:

```
make XDIR=/u/cs452/public/xdev
```

2.1 K2 Test Programs

The RPS Game and Performance measurements programs required for K2 are built using the targets `k2` and `k2-timings`.

3 Kernel Structure

The program files are structured into three parts – Kernel, User Libraries, and User. These parts each have their own directories, `kernel/`, `./` (i.e. the repository root), and `user/` respectively. In each of these directories, the header and implementation files are split into the directories `includes` and `src` respectively.

3.1 Generic Kernel Loop

The kernel is designed as a generic loop, rather than as an event handler. As part of the initialisation, the kernel performs a number of tasks:

(1) Initialise singleton data structures for task/stack allocation, scheduling, interrupt handling, and timing;

(2) Initialise the initial kernel state for context switching;

(3) Install the address of the exception vector into the `VBAR_EL1` system register;

(4) Initialise the UARTs;

(5) Allocate and initialise the nameserver;

(6) Allocate and initialise the first user task (with default entrypoint `user_main`).

(7) Allocate and initialise the idle task.

Then, on each iteration of its main loop, the kernel does the following:

(1) Remove the next task from the queue as given by the scheduling algorithm (Section 3.3.5);

(2) Context switch (Section 3.3.1) into the selected task;

(3) On execution and handling by the hardware of the `svc N` instruction or interrupt, context switch back to the kernel, returning the handler.

(4) Based on the handler, handle the system call (Section 3.3.6) or interrupt (Section 3.3.8) as described in the interface.

The kernel loop ends when all created user tasks (including the initial task) has exited, and restarts the machine.

3.2 Critical System Parameters and Limitations

The stack size is limited to `0x80000` bytes, or 512 KiB per task. The maximum number of simultaneous running tasks is 1024. These numbers were chosen because they provide a modest stack size while allowing the possibility of a large number of simultaneously running

smaller tasks, and both being powers of 2 (2^{19} and 2^{10} respectively) to allow for faster calculations. In total, this gives the program 500 MiB to work with in total task stack size, which, taking into account that we have approximately 1 Gb of program memory to work with, and leaving room for the bootloader and the program load address of `0x80000`, gives more than enough room.

The TIDs are limited by the address space of `uint16_t`, i.e. 65,535 TIDs. This is far more than the total number of tasks, while being reasonably space efficient. The TIDs are reassigned, and the algorithm for this is discussed in Section 3.3.4. The larger address space for TIDs compared to the number of simultaneous active tasks is to prevent more expensive reassignment searches from occurring frequently.

The nameserver names are limited in length to 256, since without heap allocation dynamic arrays are not really possible, nor necessary. It is incredibly unlikely that names will be longer than 256 characters, and if so, the user programmer should seriously reconsider a better naming scheme. The maximum number of names the nameserver can hold is also 256, since it is reasoned that most tasks do not need to be known, and thus being able to hold a quarter of tasks in the nameserver is more than enough.

3.3 Algorithms

3.3.1 Context Switching

Context switching is implemented in `kernel/src/context-switch-asm.S` as an assembly routine linked as a C routine. On boot, the kernel starts first and creates an entrypoint user-level task, and context switches it via the following routine:

```
void context_switch_out(task_t *curr_user_task, kernel_state *kernel_task);
```

This assembler routine pushes the addresses of the task descriptors onto the stack (important for context switching back into the kernel). It then uses `x0` to point to the beginning of the `kernel_task`, and directly stores registers `x1` to `x30` into the state using offsets of `x0`. It then lastly takes `x0` which was stored onto the stack and saves it into the state. Next, the routine has to load the saved registers of the user task. It does so again by using `x0` and `x1` as scratch registers, loading the address of `curr_user_task` into `x0` from the stack. First, it directly loads registers `x2` to `x30` using offsets of `x0`. It then loads the `pc` into `ELR_EL1`, `sp` into `SP_EL0`, and the saved `pstate` into `SPSR_EL1`, taking care to only load 32 bits. After this, it then loads registers `x1` and `x0`, in that order, since we used them as scratch registers.

Context switching in is much the same but in inverse, but makes use of the fact that after the `svc` instruction is executed, we are on the kernel stack, and thus is able to re-save the user task and restore the kernel task. However, before storing the user task, we need to push any registers we need as scratch (i.e. `x0`) onto the stack before doing anything else, and rewrite it to the task descriptor at the end.

3.3.2 Stack Allocation

User level stacks are allocated in memory following the kernel stack, with the previously discussed constraints in Section 3.2. Stack allocation is managed by the singleton `stack_alloc` structure defined in `kernel/include/stack-alloc.h`. They are allocated in linear fashion after the kernel stack, but not zeroed out to increase performance of task creation, since there was a significant lag when clearing a stack size of 512 KiB. Freed task stacks are reclaimed linearly to reduce fragmentation. It was considered to use slab allocation to better reduce fragmentation, but the overhead for managing this does not seem worth it at this time. If we see fragmentation later on when working with more tasks, this change will be made.

3.3.3 Task Allocation

The task allocation mechanism employs a sophisticated slab allocation strategy to optimise memory usage and mitigate memory fragmentation. Initially, the memory is segmented into 128 slabs, each containing 8 tasks. Whenever the free list is exhausted, a new slab is initialised, with its 8 tasks added to the free list and the slab marked as active. To track the usage, a counter is associated with each slab to monitor its active status and the number of used tasks; additionally, every task contains a variable to indicate the slab it belongs to.

When a task is added back to the freelist, it is not just added to the front of the free list. Instead, the free list is structured by slab-id, ensuring the utilisation of lower slabs first. When all tasks from a specific slab are added back to the free list, and consequently, the task usage counter for that slab resets to zero, all tasks from that slab are extracted from the free list and the slab is deallocated. If again additional memory is required, the lowest unused slab is (re)activated and its tasks are (re)integrated into the free list. This procedure ensures minimal fragmentation and optimal utilisation of available memory. This is also less expensive to do than other algorithms, such as precisely tracking allocations in a separate structure.

3.3.4 Task IDs

The task ID (TID) allocation is inspired by the algorithm UNIX uses for allocating process IDs. TIDs are allocated sequentially to tasks until they reach the maximum TID value (65,535), in which allocation returns to 300 and increases again. This is used because it is less likely that long running tasks were allocated with higher TIDs, and can obviously be adjusted in the future as we see fit to increase performance.

On each choice of TID, the kernel checks that the TID is not in use, and if so tries the next one. This is obviously worst-case expensive, but as mentioned previously, the large TID space in comparison to the number of tasks makes the search unlikely to be long. The slight increase in cost per task allocation is marginal, and if it works for UNIX, it works for us.

User tasks are allocated beginning with TID 50, since we know there will be longer running server tasks that are started by the kernel.

The Task ID algorithm is implemented in the static `task_queue_get_tid` method located in `kernel/src/task-queue.c`.

3.3.5 Scheduling

The scheduling follows a straightforward round-robin approach, housed within the task queue. Starting with the highest priority queue, it cycles through the queue to return the first task in a `STATE_READY` condition. When tasks are scheduled and have executed and context switched back to the kernel, they are rescheduled to the end of their respective priority queue. Any tasks not in ready state are skipped and maintain their position within the queue, so that when they become unblocked, they will be scheduled ahead of any tasks which may have been running while they were blocked. Should the queue be empty or only comprise of non-ready tasks, the scheduler advances to the next lower priority queue in search of ready tasks. New tasks are always added to the end of their respective priority queue.

Long running server tasks which are started in the kernel initialisation have their own highest priority, in order to enable timely service to user tasks. These tasks are excluded when checking for active user tasks, and are user tasks with special treatment.

If any of these server tasks require notifier task(s), they also can be run at a higher priority (under the server tasks), which are also excluded when checking for active user tasks. Finally, the idle task (Section 3.3.9) runs at the lowest priority without any other tasks, and is also ignored to allow for smooth exit. In essence, any server/kernel tasks are always ignored.

3.3.6 System Calls

The individual System Calls implemented for this assignment (`Create`, `Exit`, etc) are implemented in `includes/task.h`. These methods are simply a wrapper on the `syscall` procedure implemented in `includes/syscall.h`. This method takes advantage of the compiler moving call arguments into registers `x0` to `x7`, and simply calls `svc #0`, with the syscall number being stored as the first argument, and then returns (when the kernel eventually switches us back as the active task).

These arguments are preserved during the context switch, and saved in the task descriptor for processing. During this, the kernel executes code based on the syscall number in `x0`, and returns the result (if any) in `x0`, following aarch64 convention. This means that when the application is switched back, and returns to the location of the `svc` instruction, it will return the value in `x0`.

3.3.7 Message Passing

Message passing is implemented as part of three methods: `Send`, `Receive`, and `Reply`. These are defined in the header `include/msg.h`, and are implemented as system calls (Section 3.3.6. All together, these form the message passing algorithm, which allows user tasks to

communicate between each other. For our discussion we will denote T_s and T_r be the sending and receiving task respectively. For the sending and receiving sections, we will discuss the algorithm in two cases, depending on which task executes first.

First, suppose T_s executes first. When T_s executes `Send`, it blocks until after T_r has called `Reply`, which is accomplished by the kernel moving T_s's ready state from `READY` to `SEND_WAIT`. It then attaches T_s to the back of T_r's waiting queue, which is implemented as an intrusive linked list, and is separate from the task scheduler (i.e. tasks stay on the scheduler queue even if they are in a task's waiting queue). Note that the kernel can easily determine that T_s was called first by checking the ready state of T_r (i.e. in this case, T_r will be `READY`, rather than `RCV_WAIT`). Next, when T_r calls `Receive`, the kernel knows T_s went first since its waiting queue is not-empty, and pops T_s of the waiting queue. It then moves T_s to state `RPLY_WAIT`, and copies the necessary bytes (i.e. the minimum) between the given buffers, and returns the necessary information as in any system call. Note that since T_r needs to handle the receive, the kernel does not block it.

Secondly, we will consider the case for when T_r executes first, which the kernel can recognise if there are no waiting senders. In this case T_r is blocked by the kernel moving it to the `RCV_WAIT` state. Then, when T_s sends, the kernel blocks it and transitions it to state `RPLY_WAIT`, and moves the data as before. However, in this case, the kernel has to unblock T_r (i.e. ready state to `READY`) before returning the necessary parameters as previous. As mentioned previously, the kernel is able to check that T_r executed first by verifying that T_r is in `RCV_WAIT` state.

In either case, T_r processes the sent message, and needs to `Reply` to T_s. The kernel handles this by copying the appropriate number of bytes, and unblocks T_s by returning it to the `READY` state.

3.3.8 Interrupts

Interrupts are set-up and handled (i.e. acknowledgement, etc) as documented in the ARM GIC documentation, as the `kernel/include/interrupts.h` interface. In terms of the kernel receiving the interrupts, a context-switch (Section 3.3.1) is performed, and the kernel executes the interrupt handler (rather than the system call handler) based on the return code of the context switch.

The interrupt handler itself is primarily driven by the event queue (Section 3.4.5), which keeps track of tasks blocked waiting various interrupts, separately from the task scheduler queue. This priority queue allows the interrupt handler to wake-up any blocked tasks waiting for the interrupt.

For handling UART interrupts, since all the interrupt signals are logically OR'd and passed to the GIC, the handler is more complex. The handler reads the UART `MIS` register to determine which interrupt (possible multiple) fired, and thus may unblock multiple tasks waiting on different event queues. This means that tasks need to be scheduled with appropriate priorities to ensure that actions are done as expected. Unlike other interrupts (i.e. clock ticks), nothing needs to be done to trigger the interrupt, since this will occur naturally

throughout the lifespan of the program.

3.3.9 Idle Task

The idle task runs alone at the lowest priority, so that it is only ever scheduled if there are no other tasks. Timing is done with the help of the kernel, which creates the idle task and saves the TID of the idle task, and checks whenever a new task is being scheduled/unscheduled if it's TID matches that of the idle task, and if so, starts/stops a timer.

The timer is an aggregate timer, and accumulates the total amount of time between all the starts/stops, implemented in `kernel/include/stopwatch.h`.

3.3.10 Clock Server/Notifier

The Clock Service is split into three parts: Clock Server, Clock Notifier and Clock Queue, located in `include/clock-server.h`, and `include/clock-queue.h`. The Clock Server handles the three main requests `Time`, `Delay`, and `DelayUntil`, which are implemented in `include/clock.h`. The request `Time` is directly answered, while the other two requests are blocking. To block the processes, the TID of the requester and absolute clock tick are added to the clock queue (described in more detail in Section 3.4.6). Hence, `Delay` works like `DelayUntil`, but requires to get the current clock tick and adding the delay before queueing it.

On the other hand, the clock notifier constantly calls `Await`, and blocks until it gets the interrupt by the kernel, indicating that another clock tick has passed. After receiving the interrupt, it sends a 'notify' message to the clock server and waits again for the next clock tick. On 'notify' the clock server dequeues all processes which had deque time prior to the time of the clock tick.

Through sending the replies, the processes get unblocked and the scheduler will schedule them according to their priority. The separation of clock Server and clock notifier is crucial, as otherwise the clock server would be blocked by waiting for the interrupt and could not receive or process any requests.

3.4 Data Structures

3.4.1 Task Queue

Task Queue maintains one queue per priority, which keeps track of all tasks of this priority. Whenever a new task is created, it is added to the end of the respective priority queue. Removing tasks is handled by the scheduler, which decides which task is returned/removed from the queue.

The implementation is straightforward as it uses the next pointer in the task structure to maintain the queues. Task Queue itself only stores pointers to the first element (front) and

the last element (back) of each priority queue.

3.4.2 Task Descriptor

Each running task is associated to a Task Descriptor, which is used in for scheduling, handling system calls, and context switching. The Task Descriptor stores the following information:

- Registers `x0` – `x30`, `pc` (program counter), `sp` (stack pointer), and `pstate` (processor state), at time of last context switch into the kernel;
- TID (Task ID) – see Section 3.3.4;
- Pointer to parent task descriptor;
- The task's current run state (ready, running, exited, send-wait (blocked), reply-wait (blocked), receive-wait (blocked));
- The task's priority;
- An intrusive pointer to the next task in the queue, used by the Task Queue (Section 3.4.1) and the Allocator (Section 3.3.3).
- An intrusive pointer to the next task which is waiting to send to the given task, which forms a waiting queue. See Section 3.3.7.

This task descriptor structure is defined in `kernel/include/task-state.h`. This file also defines a kernel state structure, which holds the registers of the kernel for context switching.

In order to simplify context switching, all of the registers are stored in the task descriptor, rather than on the stack. This allows the kernel to maintain ownership, and easily return system call return values to the user-level programs. All the other fields of the task descriptor facilitate the data structures and algorithms which operate on the task descriptors.

3.4.3 Messages

Messages are typically passed as a C struct in byte representation. All tasks expect the first field to contain an identifier which identifies the type of structure. Some more commonly used messages are constructed as substructures with different fields, but there is no overlap in identifiers.

3.4.4 Name Server

The Nameserver is located in `include/nameserver.h`. On startup, the kernel creates the name server before any other process with TID 1. The main loop of the nameserver consists of a simple receive-process-reply loop. The nameserver saves the entries in a storage array of length 256, but maintains a bucket-hash map on top of the storage array for efficient get and insert operations.

On each register request, the server computes a hash on the name and inserts it to the back of the linked pointer list in the given hash bucket. The hashing algorithm is a simple and efficient dbj2 algorithm, which provides good distribution of hash values.

Respectively, on each `Register` request, the server hashes the requested name and maps the name to one of the 64 buckets. Each bucket contains a linked list of pointers (to the corresponding storage location in the storage array) of the entries hashed to this bucket. In most cases the algorithm only iterates a small number of entries, and if it finds an entry of the same name, it overwrites it. Otherwise, it adds it to the linked list.

The `WhoIs` method works similarly, by calculating the hash value of the given string and searching the list for a matching name.

3.4.5 Event Queue

The Event Queue is implemented in `kernel/include/event-queue.h`, and enables the interrupt handler to keep track of blocked tasks waiting for an interrupt. It is "priority" queue implemented as an array of pointers to tasks (embedded links inside the task structure), with one entry per interrupt type. This is because there is never a need to access tasks on multiple priorities, and there is no actual "priority" between interrupts. It is the most efficient to just have multiple queues. The event-queue provides methods for unblocking all or one of the waiting tasks, depending on the interrupt type.

3.4.6 Clock Queue

The Clock Queue is implemented in `include/clock-queue.h`, and enables the clock server to keep track of waiting/delayed processes. It uses a freelist of empty entry structs, composed of the tid, the clock tick and a next pointer, which is similar to previous queues. The main difference is that the clock queues adds all entries in a sorted manner, resulting in a sorted list. Respectively, on every clock tick, the queue only has to compare the absolute time stamp of the first element to the current clock tick. This enables major performance advantages, as enqueueing a process will occur very rarely compared to the clock tick.

3.4.7 Game Server

The Game Server is located in `user/include/gameserver.h`. The Game Server enables pairs of clients to play Rock, Paper, and Scissors against each other. On initialisation, the game server registers its TID with the nameserver and transitions to its main loop, waiting for clients to send signup requests. Whenever the server receives a signup request, it adds the clients to a queue, implemented as a circular array.

When two players are in the queue, and there is no active game, it dequeues two players from the buffer and sends a reply, asking for their move. The active players' status and current move are stored in a local struct to keep track of the ongoing game. After it has received

their moves, the server evaluates the outcome and waits till they either send their next move or a quit message.

On receiving a quit message from one client, it will reply at the next opportunity with a corresponding quit message to the other client. Afterwards, if possible, it dequeues two players from the buffer or waits for additional signup messages.

Furthermore, when receiving an invalid request, the server replies with an error message so the client is not trapped in a waiting state.

3.4.8 Deque

The program uses a number of queues, stacks, and reverse queues to help it accomplish many of its tasks. This includes output/input queuing, string to integer conversion, and displaying sensor data respectively. In order to cover all of these data structures with one single data structure, a `deque` (double-ended queue) is used, implemented in `collections/deque.h`. Since we have no heap (and thus cannot use a linked-list structure), the deque is implemented as a circular array (with a fixed-size maximum capacity). In order to efficiently calculate access (via modulos), the size of the deque is restricted as a power of 2 so that bit-shifting can be used, rather than the less efficient modulo operator.

3.5 I/O Servers

The `include/uart-server.h` header defines the interface for all I/O operations:

- `int Getc(int tid)`
- `int Putc(int tid, unsigned char ch)`
- `int Puts(int tid, const char *buf)`
- `int Putl(int tid, const char *buf, size_t blen)`
- `int Printf(int tid, char *fmt, ...)`

This interface is just the wrapper on `Send` calls which send the passed data (among other metadata) to the actual I/O servers, using the passed TID. We decided to split the I/O servers into two different servers: a Märklin I/O server (Section 3.5.1) and a Console I/O server (Section 3.5.2) This is because both servers have very different implementations for reading and writing bytes to the line. Furthermore, splitting these servers allow us run the Märklin server at a higher priority and prioritise its command output over the slower console output. However, these servers handle both reading and writing, as our reading is handled entirely by notifiers and interrupts (Section 3.5.1, 3.5.2). Hence, the reading is mostly decoupled from writing and thus we decided to not split the servers into read and write dedicated servers, in order to prevent the potential slower service to lower priority user tasks.

The two servers are both accessed through this interface, and it is assumed that any calling clients will know the appropriate TID for the appropriate server. Our interface differs from the given kernel description in this way, and drops the extraneous `channel` argument. Other than this, `Getc` and `Putc` are equal to the given kernel description. The other functions (`Puts`, `Putl`, `Printf`) accomplish what they usually do in the C stdlib, allowing us to pass multiple characters at a time (and even formatted) in one message send, and have similar return behaviours to `Getc`/`Putc`. We added this functionality, because it offers a significant performance improvement, as a buffer of characters can be send to I/O servers which drastically reduces the amount of messages sent.

In particular, `Printf` is essential for easy and efficient UI display in the console. However, we have to use a fixed sized buffer, as the data has to be sent in messages and shared memory is not allowed, and the size of this data must be known. Hence, the functions `Puts, Putl, Printf` are limited by `MAX_STRING_LENGTH`, which is currently set to 256.

Additionally, we introduced the function `int WaitOutputEmpty(int tid)` to the UART server. This function is implemented like the previous functions as a wrapper on a message send to the appropriate I/O server. The purpose of this function is to allow user tasks to block on the condition of the output FIFO being non-empty, and the server handles it by queuing the blocked TIDs of the task as usual and responding only when the condition is satisfied (in this case, whenever the output queue is empty). Note in this case, if the output queue is already empty on receiving the request, the calling task will be unblocked immediately. This functionality is only relevant for the Märklin server and is used in particular by the sensor data task to be interrupted when it is able to request sensor data (Section 4.4.2)

3.5.1 Märklin I/O Server

The Märklin I/O Server is composed of a server and a `MARKLIN_RX` notifier, for reading bytes, and two CTS notifiers. The server is mostly blocked on receive and handles all incoming requests from user tasks. All bytes to be written to the Märklin and all bytes read from the Märklin are buffered inside the server in a queue (Section 3.4.8).

The main part of the server is the state machine, consisting of the states `MARKLIN_READY`, `MARKLIN_CMD_SENT`, `MARKLIN_SERVER_BUSY`. When the server is in the `MARKLIN_READY` state, it will send out the next byte (if there is any in the buffer) and transition to `MARKLIN_CMD_SENT`. In this state, the server waits to receive a `CTS LO` message from a notifier and transition to `MARKLIN_SERVER_BUSY`. Once it receives the `CTS HI` message from the second notifier it transitions back to the `MARKLIN_READY` and is able to send any proceeding bytes. These three states ensure that no byte is lost in the communication with Märklin.

Further, we deviate from the classic server-notifier pattern by introducing two notifiers for the CTS signal. This is because during testing, we experienced that with certain commands, such as s88 reset byte (192), the `CTS HI` signal is reasserted very quickly after the `CTS LO` signal. However, when an interrupt occurs, the handling process is too slow, even with compiler optimisation. This is because the following must occur:

(1) The notifier needs to be unblocked and needs to send a message to the Märklin server

(2) The Märklin server needs to be scheduled and send a message back to the notifier

(3) The notifier needs to be scheduled again and call `AwaitEvent` again.

This, in some circumstances, might lead to the notifier missing the CTS High signal as it is not able to call `AwaitEvent` fast enough. To prevent these edge cases, we modified the interrupt handler to just unblock the first waiting notifier and the server to use two notifiers. Consequently, the modified pattern with blocking two notifiers at the beginning ensures there will always be a notifier ready waiting on a CTS interrupt, and thus our server can ensure it does not miss any CTS signal changes.

Lastly, the s88 sensor request commands introduces additional complexity, requiring coordination between sending and receiving bytes, and blocking sending. These commands may result in the Märklin sending 2 to 10 bytes, and during this time, no bytes can be sent to the Märklin, since the line is only half-duplex. Otherwise, this would result in data loss. To coordinate this blocking, we introduced a flag `MARKLIN_RECEIVING_SENSOR` together with a counter. Whenever the server recognises that a sensor request byte is being sent, it raises the flag and sets the counter to the amount of bytes it expects to receive.

The receiving of bytes is handled by the notifier. It calls `AwaitEvent(MARKLIN_RX)` and the interrupt handler returns bytes as it receives them.

This character is then sent to the server by the notifier, where it is buffered or sent to any waiting processes, and the counter is decremented. Once the counter reaches 0, the server can be sure that all expected bytes have been received and can enable sending to the Märklin again.

3.5.2 Console I/O Server

The Console I/O server is split into the server and the read notifier. Like the Märklin server, the server is mostly blocked in `Receive` waiting for requests. All `Put` requests are directly handled, and the server writes the passed characters directly to the console, without any buffering. Unlike the Märklin UART line, which disabled the FIFOs, it was decided the added complexity needed if disabling FIFOs for the console line was not necessary, especially since console output is not as time sensitive in comparison to sending byte to the Märklin.

We read characters from the Console FIFO via the `RTIM` interrupt, which allows us to periodically poll the Märklin UART FIFO at a less deterministic rate. The notifier waits on this interrupt and when triggered flushes the bytes present in the FIFO and sends them to the server. The server then buffers these bytes in a queue (Section 3.4.8) and is able to respond with bytes to any processes blocked with a `Getc` call, if any.

We also considered using a worker task for handling console output, since it may theoretically block if the UART FIFO is full. The advantage of such a worker task is that should this happen, the server will not block and is able to more quickly receive any income requests. This will result in a performance boost in the overall system for longer outputs, or multiple outputs in quick succession. However, this will result in an additional context switch for *every* print call, and the time required for a context switch is non-negligible, and is particularly

impactful in the case of writing single characters or short sequences of characters. For our usage, the user tasks only write very short sequences of characters to the terminal at a time, and often are not bunched together. In our testing, we found that there was no large performance hit, which led us to decide not to use a worker task. However, if we find that in the future that our console output patterns change, we will revisit this decision, but it was reasoned that this is also unlikely.

4 User Tasks

4.1 Kernel Assignment 1

The Console output of the program is as follows:

```
Created: 2
Created: 3
Task 4 parent: 1
Task 4 parent: 1
Created: 4
Task 5 parent: 1
Task 5 parent: 1
Created: 5
FirstUserTask: exiting
Task 2 parent: 0
Task 3 parent: 0
Task 2 parent: 0
Task 3 parent: 0
```

The first user task runs and created tasks 2 and 3 at a lower priority, and thus continues to run. When it creates task 4, since it is at a higher priority than it, it runs to completion first, even as it calls `Yield` since it is moved to the back of its empty priority queue. When it finishes, it returns to the first user task since it is higher priority than 2 and 3, and we see the same pattern with task 5. After 5, it returns to the first user task, which exits. Subsequent tasks have parent as 0 (i.e. the kernel) after it exits, and task 2 runs first since it was created (and thus added to the queue) first. Tasks 2 and 3 then alternate execution since they are at equal priority.

4.2 K2 - Performance Measurement

In order to measure performance as accurately as possible, we borrowed the structures used to measure time in A0 and declared them globally in order to share them between sending and receiving tasks. That way, the whichever task (i.e. sender or receiver) that went first

started the clock, and once the sender returned, the clock was stopped. This means that the timer overhead is incredibly small, since the number of instructions between accessing the clock registers and the execution is very small. The following description of the timing algorithm is adapted from Trevor's A0 documentation.

> In order to perform and report times for the documentation, the `timer_stats` structure (`user/includes/timer_stats.h`) is used to calculate the average and maximum length of time for multiple timers between start and stop. To avoid keeping a huge number of arbitrary number of times in memory, the average is calculated accumulatively, that is, the total average is calculated using the average of the previous values. This is because given values $a_1, \ldots, a_n$, we may calculate the average recursively as follows:
>
> $$\overline{a}_n = \frac{1}{n}\sum_{i=1}^{n} a_i = \frac{1}{n}\sum_{i=1}^{n-1} a_i + \frac{a_n}{n} = \frac{n-1}{n}\left(\frac{1}{n-1}\sum_{i=1}^{n-1} a_i\right) + \frac{a_n}{n} = \frac{(n-1)\overline{a}_{n-1} + a_n}{n}.$$
>
> Thus, we only keep the total number of times added and the last average to calculate the total average recursively, and save space.

In order to reduce variability, for each test situation we execute it 1000 times and take the average, using the structure above.

Optimisation Enabled?	First Process	Message Size	Time per SRR Operation (μs)
N	Sender	4	192
N	Receiver	4	194
N	Sender	64	377
N	Receiver	64	452
N	Sender	256	971
N	Receiver	256	1278
Y	Sender	4	81
Y	Receiver	4	82
Y	Sender	64	83
Y	Receiver	64	84
Y	Sender	256	100
Y	Receiver	256	106

Figure 1: A2 Performance Testing Results

The values in the `PerfTesting.csv` file are reproduced in Figure 1. From these test results, we can see first and foremost that as the message size increases, the average time per SRR operation increases approximately linearly. This makes sense because memory accesses are expensive. We can see also that enabling optimisation (i.e. compiling with the `-O3` flag) significantly reduces the operation time, and more crucially, reduces the impact on operation time due to the increase in message size, as we saw without the optimisation. This is likely

the compiler removing repeated reads/writes and using register caching to speed performance up.

Secondly, we can consistently see that when the receiver starts first, the execution time is slightly longer, and this appears to increase as the message size gets larger. Part of this is because there needs to be an extra context switch back to the receiver when the receiver sends first in order for the receiver to reply to the message.

4.3 Kernel Assignment 3

The Console output of the program is as follows:

```
Client tid 55: delay interval 10 (1)
Client tid 55: delay interval 10 (2)
Client tid 56: delay interval 23 (1)
Client tid 55: delay interval 10 (3)
Client tid 57: delay interval 33 (1)
Client tid 55: delay interval 10 (4)
Client tid 56: delay interval 23 (2)
Client tid 55: delay interval 10 (5)
Client tid 55: delay interval 10 (6)
Client tid 57: delay interval 33 (2)
Client tid 56: delay interval 23 (3)
Client tid 55: delay interval 10 (7)
Client tid 58: delay interval 71 (1)
Client tid 55: delay interval 10 (8)
Client tid 55: delay interval 10 (9)
Client tid 56: delay interval 23 (4)
Client tid 57: delay interval 33 (3)
Client tid 55: delay interval 10 (10)
Client tid 55: delay interval 10 (11)
Client tid 56: delay interval 23 (5)
Client tid 55: delay interval 10 (12)
Client tid 55: delay interval 10 (13)
Client tid 57: delay interval 33 (4)
Client tid 56: delay interval 23 (6)
Client tid 55: delay interval 10 (14)
Client tid 58: delay interval 71 (2)
Client tid 55: delay interval 10 (15)
Client tid 55: delay interval 10 (16)
Client tid 56: delay interval 23 (7)
Client tid 57: delay interval 33 (5)
Client tid 55: delay interval 10 (17)
Client tid 55: delay interval 10 (18)
Client tid 56: delay interval 23 (8)
```

```
Client tid 55: delay interval 10 (19)
Client tid 57: delay interval 33 (6)
Client tid 55: delay interval 10 (20)
Client tid 56: delay interval 23 (9)
Client tid 58: delay interval 71 (3)
Total idle time: 0:2.1 (98%)
```

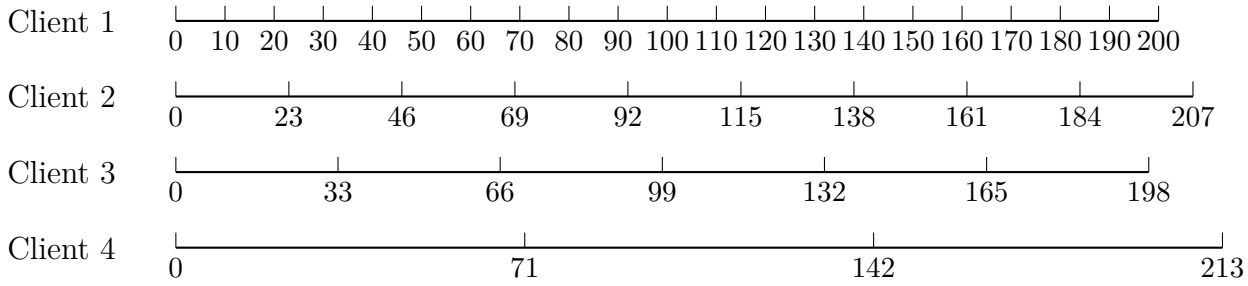
The first user task starts the clock server and the following four clock clients, with client 1 having the highest priority and client 4 the lowest:

- Client 1 (tid 55): Delay 10, No. of Delays: 20
- Client 2 (tid 56): Delay 23, No. of Delays: 9
- Client 3 (tid 57): Delay 33, No. of Delays: 6
- Client 4 (tid 58): Delay 71, No. of Delays: 3

That the user output matches the expected output can be seen with the help of the illustration below. Client 1 is the first to start and wakes up twice before Client 2 wakes up the first time. This comparison holds for the entire output sequence. It also clearly shows why client 1 is the first one to terminate and client 4 the last one.

The graphic further demonstrates that there is no clock tick where two clients wake up at the same time. Hence, the priority of the clients should have no impact in this test. However, it might have an impact if the clients would run for a longer time or if other processes are also running.

Finally the graphic shows that client 4 is finishing last.



4.4 Kernel Assignment 4

4.4.1 Overview

This program is used for controlling a Märklin 6051 controlled train set, intended to be run on the course Raspberry Pi System. It displays an accurate clock (since boot) and real-time information on the state of the track. This includes switch states, recently activated sensors, and train speeds. It also provides a CLI to set train speeds and control switch turn-outs. The commands to interact with the track are as follows:

- `tr <train number> <train speed>`: Set the speed of the specified train. The train number must be an integer in the range $[0, 99]$, and the speed must be an integer in the range $[0, 14] \cup [16, 30]$, where the latter is used to toggle the light status.
- `rv <train number>`: Change the direction of the specified train to reverse. Again, the given train number must be in the range $[0, 99]$. When run, the train will stop, before reversing at the same speed as before. Preserves the status of headlights.
- `sw <switch number> <switch direction>`: Throw the given switch in the given direction (`s`/`S` or `c`/`C`). The switch number must be valid.
- `go`: Send the 'Go' signal to the track.
- `hlt`: Send the 'Stop' signal to the track.
- `q`: Halt the program and return to boot loader. This will send any remaining commands to the track before stopping control.

The program assumes that the only trains that will operate on the tracks are numbered `1`; `2`; `24`; `47`; `54`; `58`; or `77`. However, this limitation is only to save memory and ease display, and could be easily augmented to handle an arbitrary number of trains (similar to how sensors are displayed). Moreover, integers can be specified in decimal form or in hexadecimal by prefixing `0x` (case insensitive).

4.4.2 Structure

The program is largely ported over from Trevor's A0 implementation, so any design decisions/peculiarities are documented there. The bulk of the data structures and algorithm code is untouched, other than removing the use of input/output queue parameters, and just directly sending to the appropriate I/O server (Section 3.5). The main polling-loop has been obviously removed, and replaced with three tasks which handle updating the clock, parsing/sending commands, and sending/receiving sensor data respectively. The initialisation and teardown is largely the same. All three of these tasks can be found in the file `user/src/k4/program-tasks.h`, with the user main (which starts all the necessary I/O and clock servers, and launches these primary tasks, and also performs initialisation and teardown work) being located in `user/src/k4/manual-track-controller.c`.

The `time_task_main` task is responsible for updating the displayed clock. It uses the Clock Server (Section 3.3.10) to periodically update the displayed clock, so that it does not lose ticks. The design of this task is lightweight, and its separation from others makes sense since it needs to block (rather than spin constantly) and also it holds no cohesion with any of the other tasks, making it easy and natural to separate.

The `cmd_task_main` task is responsible for handling both user input and sending the specified commands to the track. It will also update the display according to what is required (i.e. echo input, update track state, etc). It was considered to separate receiving user input from the parsing and command sending, but this was not done for three reasons. Firstly, humans are slow, and the parsing algorithm is efficient enough that there is no noticeable lag after

entering a command. Secondly, the added delay of passing the message to a child task for parsing (and context switching to it, getting it scheduled, etc) would negate any gain from separation. Finally, the Märklin I/O server buffers input/output for us, so there is very minimal blocking required.

The command task has a subtask (`reverse_task_main`) which it will create every time a reverse command is issued. This task blocks until the pre-calculated time that the train will stop and issues the commands to reverse the train and return it to its previous speed. Obviously, this task is created in conjunction with the parent task stopping the train. This design is needed because it would be unacceptable to block commands from being received while waiting for the train to stop.

Finally, `sensor_task_main` is responsible for querying and receiving sensor data. The actual coordination between any sending tasks after the query byte is sent to prevent any command bytes from being sent while sensor data is being received is handled by the Märklin I/O server. The task simply needs to wait for the output queue to be empty, and send the query byte before receiving the expected number of bytes, before trying this again. Again, like the clock display task, this separation is natural.