# CS 452 Assignment 4

## Authors

Sebastian Gaweda (20143503) and Shashwat Pratap (20638036)

## Make and Run

**checkout** - git clone https://git.uwaterloo.ca/sgaweda/cs452-rtos-aka-trains.git
**make** - cd cs452-rtos-aka-trains/k4/ && make
**run** - in gtk terminal run - load ARM/{user-name}/k4.elf; go

## Kernel Description

### Kernel Specification

Our implementation adheres to the kernel specification provided on the course website: https://www.student.cs.uwaterloo.ca/ cs452/F19/assignments/kernel.html. It also implements an additional Shutdown function which allows a program to trigger kernel shutdown (namely, the Train Controller).

### Using C++

The decision to make C++ was made very early on. We wanted to be able to leverage templating to get around some of the pitfalls we both experienced during A0 when creating data structures that use different types. We also both agreed that classes would give us an abstraction that makes it easier to organize our code.

### Kernel Class

The main component of our kernel is the Kernel class. This class runs the main loop of kernel which configures the environment and then loops through scheduling tasks, switching to them, and then handling any requests that the task may produce. It also stores the ready queue and an array of TaskDescriptors which also serves the function of setting aside stack space for our tasks.

### TaskDescriptor Class

TaskDescriptors store important task information like the tid and parent tid. Stack space for tasks is defined stored in the task descriptor along with information related to the tasks's stack state. Receive queues and message data are also stored here.

## Data-structures

We created a templated RingBuffer which can be templated with a type and the number of objects to be stored. It exposes push and pop operations and is currently used in our code as a queue.

We created a templated PriorityQueue which can be templated with a type, a number of rows, and a number of columns. The rows correspond to priorities and define how many RingBuffers the PriorityQueue has. The columns correspond to how many elements can be in each RingBuffer. The PriorityQueue's pop function selects the first item from the highest priority non-empty buffer.

## Interrupt Handling

The first major change in k3 was a massive overhaul of the SWI context switch. Global "registers" (c variables) were removed, and all important information is now stored on the user stack. The SWI context switch saves all the registers of the program that triggered the SWI, which makes it possible to use the same exit when returning from a hardware interrupt. The hardware interrupt handler is of largely the same form as the SWI context switch with some minor differences in what modes the handler switches to and what value it pushes onto the stack as a flag for the kernel. This flag is used by the kernel to determine if a hardware interrupt has occurred so that it can handle it appropriately.

## Message Passing

Our message passing implementation has each task maintain its own receive queue. Once a task has been received it is placed on a global reply queue so that any task can reply to it.

## Clock Server

One of the first things required for the Clock Server was AwaitEvent(EventId). AwaitEvent(...) is a Software Interrupt call, which takes in an EventId and based on this EventId, stores the caller on a queue. This queue contains all the tasks waiting on the same event / interrupt to occur, to continue their execution. Once the event / interrupt occurs, all the tasks on the queue are popped and their execution state is set to Ready and they are pushed on the Ready Queue. A RingBuffer is used for creating the queue to store tasks waiting on the same event.

The Clock Server creates a Clock Notifier which runs at the highest priority.

First thing the Clock Notifier does is start one of the timers (we used timer 1) and sets it to run at 508KHz and sets the initial value of the clock to 20 and makes it run in periodic mode. This gives the kernel periodic interrupts every 10 ms which is the resolution of our system clock. Once it sets this clock up, Clock Notifier goes into an infinite loop where it keeps Awaiting on Interrupts generated by timer 1. Whenever it resumes its execution after the AwaitEvent, it sends a message to the Clock Server to signify that a tick has passed. On receiving a reply it goes back to waiting on Timer Interrupts.

The Clock server runs on priority 1, and whenever it Receives a message from Clock Notifier it increments an internal variable which stores how many ticks have passed and Replies back to the Clock Notifier to let the Clock Notifier Await on a timer interrupt. It handles 3 types of inputs other than the input from Clock Notifier.

It offers functionality for Time, Delay and DelayUntil syscalls. All 3 of these functions are wrappers for Send to the Clock Server. When Clock Server gets a Send message from a task for Time, it replies with the internal count of ticks that have passed (which it tracks and increments every time it gets a Send message from the Clock Notifier).

The final 2 functions, Delay and DelayUntil are quite similar in the terms of what happens when they are called. Both Delay and DelayUntil send to the Clock Server the number of ticks they want to sleep for / sleep until. The Clock Server maintains a RingBuffer where it stores a list of the tasks which want to sleep for certain ticks. The clock server takes the ticks from the Delay and DelayUntil call and waits for the specified number of ticks to pass before replying to the task which called Delay / DelayUntil.

**UART Communication**

This addition created a lot of problems for us. Our kernel is made without optimization because turning on optimization introduces a heisenbug we have not been able to track down. Our architectural approach was to create 4 notifiers and 4 servers covering the permutations of UART number and communication type (receive or transmit). The benefit of having 4 servers is that we were able to prioritize the different communication types, giving COM1 the highest priority. It is our belief that this approach will minimize time for an instruction intended for a train to be sent to the Marklin console.

We wanted the train controller to be capable of processing commands from any source, so a server was created whose sole responsibility is processing COM2 keyboard input and passing it the train controller server as a fully formed command.

## Train Controller

The train controller takes over the task our polling loop performed in A0. Instead of polling, it is a server which waits for commands to be sent to it. It then parses the commands and performs appropriate actions.

## Sensor Server

Instead of including sensor polling as part of the Train Controller, it runs as an independent task which is configured to execute periodically. Keeping it separate ensures that the Train Controller is free to block while waiting for commands.

## Halting

When no tasks are ready, the halt program is queued in the kernel. When the kernel sees that the halt program is about to execute, it captures timing data using timer 3 running at 508KHz, and reports idle time statistics to the terminal. We wanted to have the idle task be entirely responsible for this, but making this task kernel controlled greatly simplified the logic around timing idle time.

## UI

The UI was not fully implemented, hence the input and the sensor data appear together and can get easily get jumbled. We have plans to fix this using either atomic putstr function or using a terminal server where we send all our print requests.