An Educational Kernel for the Raspberry Pi

Thomas Archbold
1602581
Department of Computer Science
University of Warwick

April 29, 2019

CS310 Third-year Project
supervised by Adam Chester

Abstract

This project presents an educational operating system for the Raspberry Pi 1 Model B+, written for the purpose of demystifying aspects of operating systems development for the hobbyist programmer, especially with regards to low-level systems programming and core features to a computer's execution. It provides a small multiprocessing kernel developed with the aim of clarity of understanding and ease of extension. Together with a simple interface to further configure the system at compile-time to modify its approaches to process scheduling, it aims to promote a practical approach to operating systems education. Future work should be aimed at increasing the operating system's usability in a real-world context, in particular with regards to user input and access to permanent storage, of which it has none. While it is limited in this sense, it provides a simple and open testbed on which to both study how key operating system concepts are implemented in practice, as well as to invite the addition of new features.

Keywords – Raspberry Pi, operating systems, kernel, process scheduling, inter-process communication

Contents

1 Introduction

Operating systems are some of the most pervasive pieces of software around, but due to their inherent complexity, their inner workings are often impenetrable to understand without specialist knowledge. While widespread access to a personal computer is nothing new, the introduction of the Raspberry Pi in recent years has rendered experimentation with computers much more affordable and hence readily available, inviting tinkering at all levels with less concern of economic loss. The Pi therefore provides an ideal platform for operating systems education – novice developers looking to get involved in writing such systems have access to a standardised set of hardware that is inexpensive both to maintain and replace, if and when things go wrong. Now seven years since its initial release, the Raspberry Pi has several official operating systems to offer, each addressing its own issue such as ease-of-installation, Internet of Things integration, or classroom management [1], with many more unofficial ones. However, there is less in the way of those written to teach concepts of the operating system itself – this project attempts to fill this gap by providing a configurable, educational operating system for the Raspberry Pi 1 Model B+. The focus is on presenting code which is simple to understand and providing clear interfaces that encourage ease of extensibility, and hence a practical, software-driven approach to learning about operating systems.

1.1 Motivation

An operating system draws together aspects from all over the field of computer science, whose development requires intimate knowledge of low-level concepts such as the computer's organisation and architecture, up to an understanding for the more abstract in designing how processes communicate or implementing filesystems. The opportunity to write one as part of this project was therefore appealing as it served not only to provide the author with experience in an area of computer science of great interest, but also the chance to unite and put into practice many of the topics learned throughout the undergraduate computer science course. One of the key motivations of this project was therefore to gain experience in low-level systems programming and interacting with real-world hardware, all the while creating a useful and entirely self-contained piece of software.

The main goal of this project is to provide an operating system for the Raspberry Pi that is capable of booting on real hardware, for both educational and hobbyist use. An important aspect of this is gaining practical experience in this unique area of software development, and so in addition to being configurable at compile-time, offering multiple approaches to process scheduling and inter-process communication, it also aims to easily-understandable and open to extension. In order to achieve this it must also implement a basic interface to manage this compile-time configuration. Also key to the project's success is the ability to boot on real hardware - while it would largely be possible to implement in a solely emulated environment with the help of a virtual machine, this project attempts to provide the additional valuable experience of taking real-world design considerations and observing their effects on live hardware, the latter of which is easily ignored if working only through an emulator.

2 Background

2.1 Relevant Material

To this end, there are a handful of modern resources for getting involved with operating systems development – a particularly useful one at the time of first carrying out research for the project's proposal was `wiki.osdev.org`, which contains information about the creation of operating systems and acts as a community for hobbyist operating system developers. However, much of the focus is on the x86 platform and past providing a brief overview of the idiosyncrasies of the Raspberry Pi as well as the code to get a barebones kernel to boot, there is little material on the specifics to get core systems working on the platform. Cambridge's *Baking Pi* [2] provides more help in this regard, with Alex Chadwick's comprehensive tutorials proving an invaluable resource for information such as accessing registers and peripherals specific to the Raspberry Pi. The project can, however, be much further extended to guide through the implementation of core operating system concepts such as memory management, the process model, inter-process communication, and filesystems. Another aspect in which this series of tutorials diverges with the goal of this project is the language in which it has been implemented – while assembly is an undeniably language in which to be competent, it is not the most easily-understandable, in stark contrast to what this project hopes to achieve. The resource which aligns most tightly with the aim of this project is [3], whose tutorials have served as an outline to how many key features of the project have been implemented.

Finally, other notable resources which are in place to teach general operating systems development are Stanford's *Pintos* [4] and Tanenbaum's MINIX operating system [5]; the former was written to accompany the university's CS140 Operating Systems course, while the latter is an illustrative operating system written alongside the book *Operating Systems: Design and Implementation*, as a means of providing concrete examples of how operating system features are implemented in practice. Helin and Renberg's *The Little Book About OS Development* [6] also serves as a guide to writing one's own operating system. The only drawback to these three is their focus on the x86 architecture, and while they are useful resources it is in concept only, given the gap which was found to quickly form from focusing on a different processor.

2.2 Why is this project worthwhile?

The project is worthwhile firstly as it provides an accessible gateway into systems programming and operating systems development. Given the relative difficulty and additional effort required to get involved with this area of software development as opposed to others, for example by reading technical reference manuals and building an intimate knowledge of the hardware with which you are working before even starting, it therefore finds its use in easing this transition and making the learning curve associated with its involvement less intimidating and more approachable. In doing this, the project is also worthwhile in that it demystifies some of the key considerations that go into operating system implementations, not only in high-level concepts such as processes, but also the low-level with notions such as memory-mapped I/O and the processor's registers. In providing this opportunity to see theory in practice, it further opens up the opportunity for experimentation and invites practical self-learning, and hopefully clarifying why existing operating systems work the way they do.

While there are similarities to be drawn between the aims of this project and those of the current background material, both looking to create a more accessible way in to operating systems development, this project addresses the gap that they leave unfilled by tackling a different architecture, as well as in approaching feature implementation in a more modular manner. Thus the project forms one more part in the ecosystem of introductory and instructional operating systems.

2.3 Useful concepts

Operating System - a program that manages a computer's hardware that acts as an intermediary between the user and said hardware [7], providing an environment in which a user can execute programs conveniently and efficiently.

Kernel - the one program running at all times throughout an operating system's execution, the kernel is often tasked with managing the most vital/recurring tasks. Other key types of programs include system programs and application programs.

Freestanding environment - typical programs are written to run in a hosted environment, meaning they has access to a C standard library and other useful runtime features. Conversely, a freestanding environment is one which uses no such pre-supplied standard, meaning a bespoke one must be supplied. Any functions we wish to use as part of the operating system we must define ourselves.

Cross-compiler - a regular compiler will generate machine-code which is specific to that on which the code has been compiled. By contrast, a cross-compiler allows us to write code on any machine and compile it for our target architecture (the architecture on which we design our code to run).

Linker - responsible for linking all object (`.o`) files generated by a compiler/assembler into a single executable (or 'library file', see [?]). It also defines the entry point and the location of the various sections (detailed in Section 3.2.2) in the final ELF file.

Exception - an event triggered when something exceptional happens during normal execution, for example hardware providing the CPU with data, a privileged action, or bad instruction.

Process - a program that has been loaded into memory and is executing.

Context Switch - the act of saving the currently executing process into memory (**state save**) followed by loading the saved state of a different process (**state restore**). It allows the CPU to switch from executing one process to executing another.

Concurrency - the ability for multiple processes to make progress seemingly simultaneously, as a result of being rapidly brought into and out of memory with respect to a policy enforced by some scheduling algorithm.

Synchronisation - the prevention of **race conditions**: when the outcome of two concurrently executing processes depends on the order in which data access took place.

Inter-process Communication - the mechanism by which processes may exchange data and information.

Peripheral - a device with a specific memory address which it may write data to and read data from. All peripherals may be described by an offset from some base address, covered in Section 4.3.

Note: Throughout this report the units kiB and MiB are used (mostly in the context of memory) to denote $2^{10} = 1024$ bytes and $(2^{10})^2 = 1,048,576$ bytes, respectively. This is to ensure clarity and avoid confusion with their SI-prefixed counterparts, namely kB and MB, which instead correspond to 10^3 and 10^6 bytes.

3 Design

3.1 Hardware

The project has been developed for the Raspberry Pi 1 Model B+. Some of the relevant hardware onboard includes:

- System-on-Chip: Broadcom BCM2835
- CPU: 700MHZ ARM1176JZF-S
- GPU: 250MHz Broadcom VideoCore IV
- SDRAM: 512MiB, shared with the GPU
- Video output: HDMI and DSI
- Storage: MicroSDHC slot
- 4x USB 2.0 ports
- 40 General Purpose Input/Output (GPIO) pins

Development was initially planned for the Raspberry Pi 2 Model B+ simply due to its availability, having been received as a gift some years prior. However, focus was switched to target the Raspberry Pi 1 Model B+ as a result of the difficulties encountered with interacting with the GPU via the mailbox interface, with the processing differing slightly and being more complex on the 2. The underlying architecture of the 1's BCM2835 chip is, however, identical to that of the BCM2836 and BCM2837 [8], used by the Raspberry Pi 2 and 3, respectively. They only differ in that the 1 uses the ARM1176JZF-S processor, as opposed to the quad-core Cortex-A7 and quad-core Cortex-A53 cluster used by these later boards, in addition to the 512MiB extra available to them. Therefore, the choice between specific models for which to develop would have made little difference to the outcome of the project, and indeed much of the code is transferable. As has been discussed, this standard set of hardware was desirable for the project's aims, rendering the operating system more widely accessible and for less effort.

3.1.1 The Raspberry Pi's boot process

The decision to work with the Raspberry Pi in particular, as opposed to a more open-ended PC setup, is not only due to the relative lack of material available for operating systems development on the ARM architecture, but also as a result of the much simpler boot process in contrast to other hardware, details of which were found from [9]. Booting is handled almost entirely by the Pi's system-on-chip, thus does not require the writing of a custom bootloader. Instead, it relies on closed-source proprietary firmware

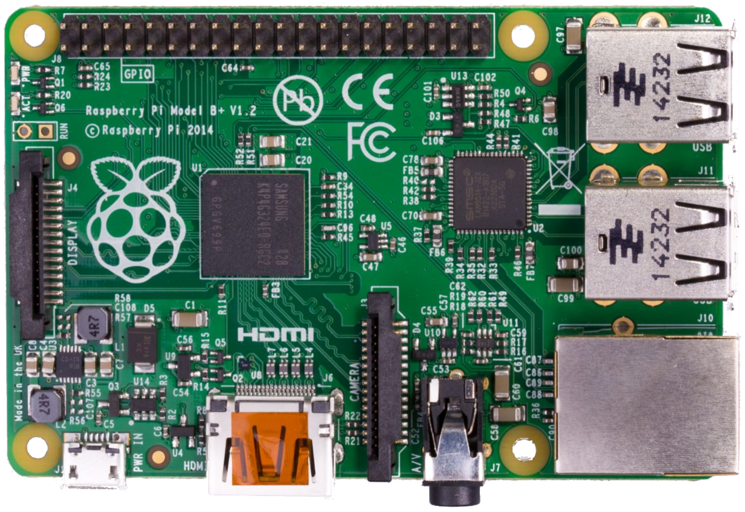

Figure 1: Raspberry Pi 1 Model B+

programmed into the SoC processor which may not be modified. The necessary files can be acquired by either downloading them from [10], or by downloading an existing operating system for the Pi and using the files that it provides (since it still requires the same firmware).

During system boot, the ARM CPU does not act as the main CPU, but rather as a coprocessor to the VideoCore GPU [11]. When the system is first powered on, the ARM CPU is halted and SDRAM is disabled. Control is passed to the GPU, whose responsibility it is to execute the bootloader. The bootloader itself is divided into three stages: the first stage, stored in ROM on the system-on-chip; the second stage, `bootcode.bin`, and the third stage, `start.elf`. The first mounts the FAT32 boot partition on the SD card to enable execution of the second-stage bootloader, and then loads this into the L2 cache to run it. Control is then passed to the second-stage bootloader, which enables SDRAM and loads `start.elf` for execution. This final stage allows the GPU to start up the CPU. An additional file, `fixup.dat`, is used to configure the shared SDRAM partition between the two processors. The GPU firmware reads the files `config.txt` and `cmdline.txt` to load the kernel image, then releases the CPU from reset and transfers control to it to begin executing the kernel.

After an operating system is loaded, the code on the GPU is not unloaded, but rather runs its own simple operating system, the VideoCore Operating System (VCOS) [12]. This can be used to communicate with the services provided by the GPU (for example, providing a framebuffer), using the mailbox peripheral and ARM CPU interrupts, which the GPU is capable of producing. The GPU is not only in charge of graphical functions as, for example, it also controls the system timer and audio – in this way it is therefore more akin to a regular PC's BIOS.

3.2 Resources & Environment

The project was developed on various x86 machines running the Linux kernel version 4.16 onwards. Since the target environment, the Raspberry Pi 1 Model B+, runs on an ARM CPU, the target architecture is therefore different to that of the development machines. Therefore, a cross-compiler is required in order to compile the code for the target machine. Available on the ARM developer website is the GNU Embedded Toolchain [13]. Conveniently, this suite of tools is available from Arch Linux's package manager, Pacman [14], and this is the version that has been used to develop the project.

3.2.1 Technical Documentation

With no prior experience working in assembly and, in particular, how to use ARM assembly in conjunction with the C programming language, it was necessary to become better acquainted with this environment, including becoming familiar with the instructions available and the calling conventions that must be followed (for example, returning the result of an assembly procedure call in register 0). This information was gathered by reading through both official and unofficial documentation on the ARM environment in general and the specifics of working with the Pi. In particular, the following were the main resources of technical documentation used throughout the project:

- *ARM Architecture Reference Manual* [15]
- *ARM1176JZF-S Technical Reference Manual* [16]
- *ARM Developer Suite Assembler Guide* [17]
- *ARM Cortex-A Series: Programmer's Guide* [18]
- *Broadcom BCM2835 ARM Peripherals Manual* [19]

The technical and architecture reference manuals provide in-depth information for such concepts as the programmer's model or interaction with different pieces of hardware on board the Pi. They also include architectural information regarding the Pi's three standard coprocessor extensions: the system control processor (coprocessor 15), the Vector Floating-Point unit (coprocessors 10 and 11), and the debug architecture interface (coprocessor 14). The online guide at [17] provides comprehensive coverage of the instructions to use when using assembly to program on the ARM CPU, and used in conjunction with the programmer's guide provides an understanding of when and how to use certain instructions. Although not written for the processor used by the Raspberry Pi 1, instead written for the CPU used by the 2 and 3, the programmer's guide provided further examples of various instructions' use-cases, and its utility continued even when focus shifted from the Raspberry Pi 2 to the Raspberry Pi 1.

On the Raspberry Pi are various peripherals, such as the Universal Asynchronous Receiver/Transmitter (UART), system and ARM timers, or the interrupt controller. Information regarding the layout of their registers and the memory addresses from and to which to read and write was obtained from the peripherals manual. Since the underlying architecture behind the BCM2835 and BCM2836/7 is largely the same, the manual was helpful both in development for the Raspberry 1 and 2, with the BCM2836 [20] providing some extra processor specifics in the case of the latter.

3.2.2 System V ABI

Of particular use within GNU's suite of tools is the cross-compiler for `arm-none-eabi`, which provides a toolchain to target the System V ABI (Application Binary Interface). This is a set of specifications that detail the calling conventions, object and executable file formats, dynamic linking semantics, and more, for systems complying with the System V Interface Definition, of which the Raspberry Pi is one. For example, it defines the Executable and Linkable Format, or ELF, which is a format for storing programs or fragments of programs on disk that are generated as a result of compiling and linking. Each ELF file is divided into sections, specifically:

- `.text` - executable code
- `.data` - global variables which are uninitialised at compile-time
- `.rodata` - read-only data i.e. global constants
- `.bss` - uninitialised global variables

These are covered in more depth in the discussion of the linking process in section 4.1.1. Additionally the SysV ABI defines the `.comment`, `.note`, `.stab`, and `.stabstr` sections for compiler and linker toolchain comments and debugging information.

3.2.3 ARM environment

The ARM1176JZF-S processor implements the ARM11 ARMv6 architecture [16], supporting both the ARM and Thumb instruction sets[1]. The processor contains 33 general-purpose 32-bit registers and 7 dedicated 32-bit registers, 16 of which are accessible for general use at any one time in the ARM state. These are as follows:

[1]A subset of the most commonly-used 32-bit ARM instructions. Each Thumb instruction is 16 bits long, and has a corresponding 32-bit ARM instruction with an equivalent effect on the processor model. While not useful and out-of-scope for this project, the instruction sets can be easily switched between to enable the programmer to optimize for either performance or code density as they see fit.

- `r15` - Program Counter
- `r14` - Link Register
- `r13` - Stack Pointer
- `r12` - Intra-Procedure-Call Scratch Register
- `r4-r11` - local variables to a function
- `r0-r3` - arguments passed to a function, and the returned result

The Current Program Status Register (CPSR) contains code flags, status bits, and current mode bits. Another register, the Saved Program Status Register (SPSR), is similar to the CPSR but is only available in privileged modes (see Section 4.5.2. This contains the condition code flags, status bits, and current mode bits saved as a result of the exception which prompted the processor to enter the current mode.

The architecture asserts a full descending stack: full, meaning the stack pointer points to the topmost entry in the stack[2]; and descending, meaning the stack grows downwards, starting from a high memory address and progressing to lower addresses as items are pushed.

As a final note on correctly interacting with the ARM environment, any procedure calls must preserve the contents of registers 4 to 11 and the stack pointer. Further, subroutines calling other subroutines must save the return address (found in the link register) to the stack before calling that subroutine.

3.2.4 Tools

The following is a summary of the tools used to develop the project:

Languages	C, ARM assembly
Cross-compiler toolchain	GNU Embedded Toolchain (`arm-none-eabi-*`)
Build automation	GNU Make
Version control	Git, hosted remotely on Github
Emulation	QEMU
Documentation	Doxygen

Table 1: Tools used by the project

The project uses the C language both due to its familiarity as well as its ease-of-setup on the embedded environment, however there are times in such an environment, and especially for operating systems development, that assembly is more suitable [21], and was thus opted for instead. As already discussed, the GNU Embedded Toolchain has been used to target the ARM architecture for which the project is built.

The GNU Embedded Toolchain consists of several utilities, all of which are used at various stages in the project's development: `arm-none-eabi-gcc` is responsible for compiling the C and ARM assembly files into object files; `arm-none-eabi-ld` is an embedded platform-independent linker, used to link multiple object files into a single `.elf` file; `arm-none-eabi-objcopy` converts the resultant `.elf` file into a binary (system executable, or `.img`) file; `arm-none-eabi-objdump` provides helpful debugging information about the final executable file (for example, used as a disassembler it presents the kernel image in assembly form); and `arm-none-eabi-gdb` provides the familiar interface of the GNU debugger.

As the project increased in complexity, especially with the addition of multiple source files, GNU's Make utility was used to automate the build process. This involved becoming familiar with concepts regarding rules, with GNU's online manual page [22] and [23] being of particular help.

Throughout its early stages, the project operated solely in an emulated environment, simply as it provided quicker feedback with respect to testing, which is already limited and slow in such an environment. For this purpose QEMU was consequently used. However, due to its lack of simulation of a system timer, an important aspect when programming interrupts and process scheduling, focus had to eventually be shifted towards operating on real hardware, limiting QEMU's influence on the project as a whole.

[2]Contrasted to empty, which points to the next free location, i.e. the address at which the next item will be stored.

3.3 System Overview

The project presents an operating which is capable of booting on a Raspberry Pi 1 Model B+ and running (with reduced functionality) on an environment emulating the Raspberry Pi 2 Model B, and initialises a C runtime environment. It provides a mechanism for memory management, first in the form of determining the total memory available to the system, and then in dividing this up into 4kiB pages. A dynamic memory allocator is set up in the form of a 1MiB portion of "heap" memory, and provides an interface similar to the C standard library's `malloc()` and `free()` in order to manage this.

Visual feedback is provided in the form of output to HDMI and, in the virtual environment, output to serial via the UART peripheral. The mechanism for communication with the GPU is set up and managed using the mailbox peripheral, and output via HDMI is achieved by requesting and providing relevant data (dimensions, bits per pixel, etc.) to the framebuffer. User interaction is so far only achieved in the virtual environment, again doing so via UART, with real-world interaction via USB proving to be a significant source of challenge throughout development.

The project sets up interrupts and exceptions by providing various exception handlers. This is with the main aim of interacting with the system timer, a peripheral capable of generating interrupts every set amount of time, for use in process scheduling. It also provides a somewhat intuitive interface for registering new "types" of interrupts (further covered in Section 4.5.4). Processes are implemented and stored using the standard job- and ready-queues, and a simple interface for creating new threads of execution is provided.

With process scheduling being one of the main systems designed for ease of understanding and extensibility, the project provides a simple interface for using and programming custom process schedulers, with all code required contained entirely in one file dedicated to such a task. Taking into account the added complexity brought on by concurrency, synchronisation is achieved by implementing both spinlocks and mutual exclusion locks.

A simple and somewhat rudimentary form of inter-process communication is provided, making use of the "shared memory" model. User interaction via USB keyboard is attempted and provided as a compile-time option, and makes use of a statically-linked library provided by Rene Stange's project *USPi* [24]. While not an initial requirement, the system provides the ability to enable the Memory Management Unit, allowing the programmer to define their own mapping from virtual to physical memory and control which memory is bufferable and cacheable.

The project also provides a means of configuring various options at compile-time through sending directives via the Makefile, resulting in a clean and simple interface to change between various systems. At present, the system may be configured for running on the Raspberry Pi 1 or 2, using Round-Robin or First Come First Served Scheduling, and which form of inter-process communication to use (for which only shared memory is currently provided). While there is not much choice currently, this system is relatively simple and easy to extend as more features are developed.

Finally, since the ability to demystify the codebase of an operating system is an important aspect of the project, it has been designed to be compatible with Doxygen [25] in order to generate documentation for the project in a simple collection of HTML files. It provides similar functionality to Oracle's Javadoc [26].

Future work will be concentrated on achieving user-interaction via USB keyboard and using this to implement a shell/command interpreter. A final important system to implement is filesystems and permanent storage, for which an interface to the External Mass Media Controller (EMMC) must be written, to enable interaction with the SD card.

3.4 Project management

3.4.1 Development methodology

The project has been developed in a waterfall-style approach – this has been well-suited to the rigidity of system dependencies, especially in early versions of the operating system. For example, it was necessary to write the boot code and set up the C environment before writing a interrupt handlers before implementing process scheduling, and any deviation from this order would have made little sense. This inflexibility allowed for such core features to be planned and designed in advance, and meant that focus only progressed to another feature once they had been implemented successfully. Once these lower-level systems were in place, the project opened up more, particularly after having completed development of processes; from this point focus at any point could be concentrated on features such as process scheduling, inter-process communication, or user interaction. As such, development became more agile, and the decision to pursue development on certain features was taken by assessing their importance to the project's success in relation to the other possible features. Throughout the course of the project, goals have been set and achieved in an incremental manner; once deciding upon the next system to implement, this involved breaking it down into appropriate subtasks and completing each in turn. An example of this has been the development of memory management, first requiring the parsing of atags, followed by creating and maintaining a list of page metadata, and finally using these to write the dynamic memory allocator.

3.4.2 Organisation

From the very start of the project, the effective use of version control software has been vital to its success. Integrating with Git and hosting it remotely on Github occurred early on and has been a constant consideration throughout. Since testing on real hardware is so important to determining whether or not an embedded system truly works, the ability to keep track of all previous builds has been useful. In particular, when features do not work, and especially if their failure interferes with other systems, being able to revert to the previous working version has helped not only in getting the project back to a functional state, but also in investigating and understanding precisely why certain attempts broke the build. Github also provides several useful features to track the evolution of the codebase, from detailing code frequency to monitoring branches. Figure 2 highlights Github's consistent aid throughout the project, in the form of weekly commits over the span of the its development.

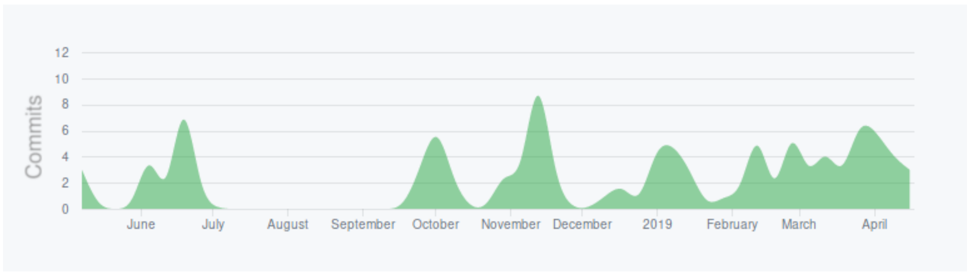

Figure 2: Weekly commits to the Github repository (source: repository's "Insights" page)

Progress was initially considered in the project specification, for which a preliminary schedule was devised. After two months of development, this was then revised in the progress report. A copy of this version is provided in Appendix A, and an evaluation of how effective and accurate these timetabling estimates were is discussed in Section 6. In order to monitor progress, regular meetings were held with the project supervisor, Adam Chester, during which recent achievements and current issues were discussed, with guidance given in the case of the latter. These further aided in consolidating lessons learned and milestones passed, when otherwise it might be easy to lose track – not vital, but helpful in tracking the project's progression. During Term 1, these meetings were held fortnightly as work on the project was less of a focus in the light of the higher course load regarding other modules. The frequency of meetings increased to weekly the following term as more time became available and more challenging and abstract aspects of the project presented themselves.

3.4.3 Testing & Debugging

The project was tested throughout development, and with particular frequency once able to boot on real hardware. This means that, once a subtask had been completed, the entire system was compiled and loaded onto hardware, with specific unit tests written each time to ensure the feature's correct operation. Also important was integration testing, so as to ensure a working implementation of one feature did not compromise the correct functioning of another. This was the case when enabling the MMU, having worked in isolation from other components, but interfering with the return addresses of other important functions when used in the wrong place or memory addresses in-use and to which it should not have had access – this only became clear as a result of integration testing.

There is little infrastructure to debug low-level software of this type, so much had to be done manually. Specifically, this involved inspecting the `.map` and `.list` files and using the Linux command-line utility **`nm`** [27] to display information about the generated executable files. Furthermore, prior to successfully sending output to a real screen via HDMI, there was no way to gather textual information about the system during execution. The only other useful way to provide debugging information during execution was interacting with the two programmable LEDs on the Pi, namely the ACT and PWR LEDs, for example causing them to blink at different points and in different patterns. Soon after successfully displaying to HDMI, it was important to develop a working version of `printf` to display debugging information in a flexible manner.

3.4.4 Legal, Ethical, and Social Issues

All software being used to develop is available under the GNU General Public License, which grants permission for commercial and private use, as well as modification and distribution. Most software used as a study resource is similarly available under the GNU General Public License, while several others are available under the MIT license, the main difference being software distributed under the former must be made open-source, while this is not true of the latter. Throughout the project's development, informal feedback has been sought from friends and colleagues, particularly for non-functional aspects of the software, such as usability and overall polish. Due to the informal nature of the feedback, however, ethical, social, and legal considerations need not apply.

4 Implementation

The following section largely documents the system as it was developed chronologically, and is not representative of the order in which systems are initialised in the final piece of software. This better highlights the dependencies between each discrete stage in development and reinforces the plan-driven approach to the project as a whole.

4.1 System initialisation

There are two main files which handle the entirety of system startup, namely `linker.ld` and `boot.S`, the former responsible for defining the layout of the final executable we will be producing, and the latter handling the setup of the C runtime environment and passing control to the main kernel function.

4.1.1 Linker Script

In order to create a kernel, or any program for that matter, we must link all compiled object files into a single executable. For user-space programs, that is, programs written to run in an established operating system, there are default scripts provided by the compiler which do this. Since the operating system on which we will run the kernel does not yet exist, we must create a linker script ourselves.

We begin by declaring that the symbol `_start` is the entry point for our entire program (this is usually `main` for user-space programs), meaning execution will jump to this address when the system starts. We first declare the symbols `__start` and `__text_start` at address `0x8000`, which informs the bootloader where to place our kernel image. Addresses up to `0x8000` will be reserved for the stack. After this we must declare one by one each of the major sections which will be used in the ELF file, the first of which is the `.text` section, containing executable code. The code from `boot.S` is placed in the `.text.boot` section, and we use the `KEEP` directive to inform the linker not to optimise the code in that section – we want it running exactly what is written there. We declare the sections `.rodata`, `.data`, and `.bss` similarly, each time declaring global symbols for them should the need to use them as variables arise (as is the case, for example, when placing the metadata for pages directly after the kernel image, see Section 4.2.2). We also use the directive `ALIGN` to define a page size for the kernel image – specifically, this sets the current address to the next available address that is divisible by the page size we set, 4096 bytes.

4.1.2 Initialising the C runtime

With our kernel image sections defined, we can actually write the code that will boot into our kernel, contained in the file `boot.S`. We begin by declaring that this is to be placed in the `.text.boot` section (as used by the bootloader) and we make our start symbol visible from any file using the `.global` directive – this is required for the linker to see which location to jump to at system startup.

The project was initially written for the Raspberry Pi 2, containing the quad-core Cortex-A7 CPU. However, since multicore programming entails extra difficulty even in a familiar user-space environment, it was always designed to only use one of these cores. The following code remains included in the project, wrapped in a test to determine which processor we have compiled for, and sends three of the four cores to hang. We will discuss it now to introduce the idea of the ARM coprocessor:

```
_start:
    mrc p15, #0, r1, c0, c0, #5
    and r1, r1, #3
    cmp r1, #0
    bne halt
```

Listing 1: Code to halt three of the four cores

The `mrc` and `mcr` instructions allows for interaction with the system control processor, and denote "move data from coprocessor to ARM register" and "move data from ARM register to coprocessor" respectively. The purpose of this is to control and provide status information for the functions implemented on the ARM1176JZF-S processor [16, pg. 3-2], and exposes optional[3] additional functionality that is not provided by the core instruction set. Among others, these functions include:

- System control and configuration
- MMU control and configuration
- Cache control and configuration
- Direct Memory Access (DMA) control
- System performance monitoring

As it stands, the project only makes use of the first two functions. The code block itself reads the Multiprocessor Affinity Register [28] for the identifier of the current CPU it is testing, and if neither of the two lower bits returned are 1, then the core is sent to halt indefinitely in a low-power state. Using the system control processor is simply a case of finding the desired functionality from [16, pg. 3-14] and copying the `mrc` or `mcr` instruction it specifies.

With just a single core to develop for, the next step is to set up the stack pointer at address `0x8000`. The program counter for the kernel starts at address `0x8000` and grows upwards, and, since the stack grows downwards, it is safe to place it at this address without it interfering with the kernel. Next the start and end of the Basic Service Set (BSS) is loaded into registers – recall that this is where statically allocated global variables are stored. Since the C standard requires uninitialised global variables to be zeroed, we must zero out (that is, store the value 0) in the registers between the addresses of `__bss_start` and `__bss_end`.

The final responsibility of `boot.S` is to transfer control to our kernel entry point, `kernel_main`, by loading the address of this symbol into the program counter. Overall, this small portion of the codebase initialises a minimum C environment, meaning the stack is initialised and the BSS segment zeroed before we pass control to our custom kernel. Note that the use of registers 0, 1, and 2 is avoided as the bootloader uses these to pass system parameters and hardware information to the main function[4] at runtime.

4.2 Organising memory

4.2.1 Atags

One important piece of data passed by the bootloader to the main kernel function is the address of the "atags" – this is a list of tags which describe various system parameters such as total system memory, the initial ramdisk, and command-line parameters to pass, and is passed through address `0x100` on the Raspberry Pi 1. Throughout the execution of the operating system, computations will require memory and since the project functions entirely in kernel-space, any and all memory is available to use. To impose some structure on the memory and manage it effectively, however, we organise it into 4kiB pages, allowing for equal-sized blocks of memory, which are neither insignificantly nor prohibitively large, to be allocated and tracked. Therefore, the most important piece of information among the atags is the total system memory, which the project uses in its implementation of a dynamic memory allocator.

In order to parse this list of tags, we need the kernel to understand the layout of the data it is being passed. In particular, each tag consists of two values: an unsigned 32-bit integer denoting the length of the tag (in 32-bit words), and the tag itself, details of which are found at [29]. This provides information

[3]ARM only provides specifications for processors; it is down to hardware manufacturers to implement their designs.

[4]Namely, the device from which the system was booted, the ARM Linux Machine Type (`0xc42` for the Pi), and the address of the atags, which contain more important information such as available memory.

concerning the memory tag itself, namely that it contains two unsigned 32-bit integers describing memory size and start address, in this order. Therefore, we define the C structure `atag_mem` to easily access these two fields once we come across the memory tag. Thus, to determine the total memory available we iterate over the atags list until we find the memory tag, cast this to our `atag_mem` structure, and consequently return the field denoting the size of system memory.

4.2.2 Pages

The total number of pages is the total size of memory divided by the page size, which we have set to 4kiB. In order to effectively manage each page, we create a list containing metadata for each page: this metadata includes, for example, the virtual address to which it maps, along with bit-field flags detailing whether the page is allocated, if it available for sharing, if it is a kernel page (for when user-space is implemented), and whether it is a regular page or a page on the heap. We use the global symbol `__end`, declared in the linker script, to place the page metadata list directly after the end of the kernel image.

To allocate a page, we need of knowledge of which ones are free to do so, for which we create a linked list of all such pages. Then all that is needed is to return a pointer to its location in memory and zero it (for security), modify the appropriate flags in the page array, and remove it from our free pages list. When a page's use is no longer required, it makes sense to make it available for future allocations, for example when a process has finished executing (see Section 4.7.3). Thus, to free a page we use its physical address to index into the array containing all pages, set the `allocated` to 0, and add it back to the list of free pages.

4.2.3 Dynamic memory allocator

To allocate memory dynamically, and less restrictively than entire pages at a time, we reserve a 1MiB portion of memory directly after the page metadata for heap memory, which is used to allocate segments of memory instead, with a similar interface to the C standard library's `malloc` and `free`. This choice of 1MiB is fairly arbitrary, but is not too restrictive for most uses of dynamic memory and does not use too large a portion of memory that will eventually be used by user-space.

Memory allocation is implemented by first defining for each allocation a header containing the allocation size, in bytes, and whether it has been allocated, and then using these headers to form a linked list. Thus, in order to grant an allocation we must find a header that is at least the requested number of bytes in size, and not currently in use. If the best-fitting segment, that is, the segment with the smallest size that satisfies the request, is relatively large compared to what was requested, we split it so that we may use what is effectively "left over" to satisfy some future request.

When freeing memory allocations, we mark the segment as unallocated by modifying the bit in the allocation header and merge adjacent free segments in the linked list into one, thus decreasing the likelihood of internal fragmentation – unused or unusable memory within the segment [7, pg. 363]. Coalescing adjacent free segments to the left is shown below; the process is identical for coalescing to the right, except that it modifies the pointer to the `next` segment header instead.

```c
/* merge segments to the left */
while (seg->prev != NULL && !seg->prev->allocated) {
    seg->prev->next = seg->next;
    seg->prev->segment_size += seg->segment_size;
    seg = seg->prev;
}
```

Listing 2: Coalescing heap segments to the left

When initialising the heap at system startup, we must first allocate the appropriate number of pages and mark them as kernel heap pages, then initialise the linked list of segments by declaring a single segment 1MiB in size. Although the total heap size remains the same, it will become split into increasingly many segments, all forming a linked list, as allocation requests are satisfied.

4.3 Serial Output

An operating system would be useless without an efficient means of interacting with it during execution, so the next task to present itself was that of interfacing with the Universal Asynchronous Receiver/-

Offset	Register	Size
`0x00`	Data Register	32
`0x04`	RSRECR	32
`0x18`	Flag Register	32
`0x20`	Unused	32
`0x24`	Integer Baud Rate Divisor	32
`0x28`	Fractional Baud Rate Divisor	32
`0x2c`	Line Control Register	32
`0x30`	Control Register	32
`0x34`	Interrupt FIFO Level Select Register	32
`0x38`	Interrupt Mask Set Clear Register	32
`0x3c`	Raw Interrupt Status Register	32
`0x40`	Masked Interrupt Status Register	32
`0x44`	Interrupt Clear Register	32
`0x48`	DMA Control Register	32
`0x80`	Test Control Register	32
`0x84`	Integration Test Input Register	32
`0x88`	Integration Test Output Register	32
`0x8c`	Test Data Register	32

Table 2: Each register is described by an offset from the UART base address, `0x20201000`.

Transmitter, a peripheral on the Pi, which is capable of serial[5] communication. While output would evolve to be via HDMI, it was simpler to initialise this when working within QEMU, and also prompted for an intuitive interface for GPIO to be written, which would again become useful when debugging with HDMI output.

4.3.1 Peripherals

As mentioned, a peripheral is a device with a specific address from and to which it may read and write data, and all interactions with peripherals make use of Memory Mapped I/O (MMIO) to do so on the Raspberry Pi – the process of performing I/O by reading from and writing to predefined memory address. Moreover, each peripheral may be described by an offset from the Raspberry Pi's Peripheral Base Address. This varies for different models of the Pi, but on the Raspberry Pi 1 Model B+, this is located at address `0x20000000`, and the physical address for peripherals span from this base address up to `0x20ffffff`. The peripheral itself contains a collection of registers which may be read and written, and these are defined at offsets from that specific peripheral's base address (see Table 2 for clarification).

There are several steps to initialise the UART for use on the Pi, and involves setting various configuration flags. The following steps are performed in order in `uart_init()` in `gpio.c`:

1. disable all aspects of the UART
2. disable all GPIO pins
3. disable pins 14 and 15
4. clear all pending interrupts
5. set up the baud rate of the connection
6. enable FIFOs and set word length to 8 bits
7. disable all interrupts
8. enable the UART hardware, and the ability to transmit and receive data

To send a byte to the data register, we wait until the FIFO (the data structure by which the UART sends and stores information) is not full and write the byte to the data register. Receiving a byte is done by waiting until the FIFO is non-empty and returns whatever is in the data register. Thus we have serial input and output, and may use this to implement `putc` and `getc` respectively. When running this build on QEMU, passing the `-serial` flag enables the sending and receiving of serial output via the host

[5]One bit at a time

computer, thus we can specify `stdin` to be able to send character input via the host system's keyboard. This enables us complete interactivity with the kernel in the emulated environment.

4.4 Interacting with the GPU

While QEMU was a powerful tool to allow for quick development and testing of core low-level features, as discussed its lack of simulation of a system timer required that the project eventually be moved to real hardware to tackle core features such as processes and interrupts. Furthermore, the Pi comes without a dedicated serial port, therefore without a TTL-to-USB adapter we are unable to use the UART to output to a real screen. Instead, it is much more natural to use the onboard HDMI port to output to a display, and for this we require communicating with the GPU.

Displaying anything to the screen via the HDMI requires the use of a framebuffer; this is a piece of memory which is shared between the CPU and GPU. The CPU writes RGB pixels to the buffer and the GPU reads from it to render the pixels to whichever output device is connected. We must first request a framebuffer from the GPU, and this interaction takes place over the mailbox peripheral.

4.4.1 Mailbox Peripheral

The mailbox peripheral is a peripheral, just like the UART, which facilitates communication between the CPU and GPU, more specifically the VideoCore Operating System, and starts at offset `0xb880` [30]. It contains a read register, located at offset `0x0` from the base, which holds messages sent by the GPU; a status register, located at offset `0x18`, which is used to signify whether the read register is empty or full; and a write register, located at offset `0x20`, which the CPU can use to send data to the GPU. Since the mailbox peripheral can also be used to send and receive information about multiple systems, such as power management, audio, and the on-board camera, the peripheral requires communication through certain channels. This is simply a number that specifies the meaning behind the data being sent; for the framebuffer, we communicate via channel 1.

Crucially, the process of communicating with the GPU via the mailbox peripheral differs depending on the model of Raspberry Pi being used; since the project was initially intended to run on the Pi 2 Model B, an interface was written for this version, however since it is a much simpler process on the Pi 1 Model B+, after some weeks debugging and making little progress, the decision to switch the target platform was switched at this point. Within the source code are both methods, and both are guarded by macros (see Section **??** on Makefile directives) which will prevent compilation of the wrong one.

4.4.2 Initialising the framebuffer

We therefore require two functions to initialise the framebuffer: one to send the framebuffer request to the GPU, and one to read its response. These are implemented as `mailbox_send()` and `mailbox_read()` respectively. Both require as an argument the channel number through which to communicate, and the former requires the additional parameter of what to send. Since we are requesting a framebuffer, we map one out in a C structure containing all the fields necessary to fully describe this area of memory, such width, height, and the maximum number of columns and rows (for text). Listing 3 shows the entire structure used to describe the framebuffer in the operating system. Two important pieces of information we must set are the depth and the pitch of the framebuffer. The framebuffer's depth is the number of bits in every pixel; we are using 8 bits for each of the red, green, and blue component, giving a total depth of 24. The pitch is the number of bytes per row of the screen. We can then calculate the number of pixels per row as $n = p * \frac{8}{d}$, where p and d are the values for pitch and depth, respectively. To calculate memory address a of a pixel located at coordinate (x, y) we have $a = p * y + (\frac{d}{8})x$. To initialise the framebuffer, we define the values we wish to set in a temporary structure representing the request, send this request to the GPU through the mailbox via the framebuffer channel, and, if successful, write these values to the framebuffer.

```
struct framebuffer {
    uint32_t width;
    uint32_t height;
    uint32_t pitch;
    void     *buffer;
    uint32_t bufsize;
    uint32_t max_col;
    uint32_t max_row;
    uint32_t col;
    uint32_t row;
};
```

Listing 3: Structure representing the framebuffer

4.4.3 Writing to the screen

The field `buffer` in the `struct framebuffer` is the area of memory to which we may write in order to display information on the screen, and in particular contains writing to it directly modifies the pixels displayed. Using the previous formula to determine the correct memory address to write for a given pixel, we implement this functionality as follows:

```
void write_pixel(uint32_t x, uint32_t y, const struct pixel *color) {
    uint8_t *loc = fb_info.buffer + y * fb_info.pitch + x *
       BYTES_PER_PIXEL;
    memcpy(loc, color, BYTES_PER_PIXEL);
}
```

Listing 4: Implementation of `write_pixel`

This uses our own implementation of the C standard library's `memcpy` along with another structure `struct pixel`, which is simply three 8-bit unsigned integers describing the pixel's red, green, and blue values. For illustration, when the framebuffer is initialised we wish to clear the screen by writing black pixels to each location, resulting in 307,200 calls to `write_pixel()` for a 640×480 screen.

With the ability to write arbitrary pixels on the screen, we may define a now define a font in order to write each individual pixel constituting a character, printing a character to the screen. For this purpose we use Daniel Hepper's 8×8 bitmap font [31], which simply defines for each character the pixels which are filled and unfilled. We also take inspiration from his rendering function, allowing us to determine whether the n^{th} pixel is filled by right-shifting by n, using `write_pixel()` in any case to modify the current pixel accordingly.

In the function `gpu_putc()`, we use this method to print only the printable ASCII characters (characters whose code is between `0x21` and `0x7e`). We also maintain two variables, `row` and `col`, which allows us to print to the correct 'character coordinate', to avoid having characters overlap. Each time a character is written, printable or not, we increment the `col` counter until we reach the maximum width of the display, at which point we increment the `row` counter and set `col` to zero. There are several special cases:

- `\t` - increments `col` by 4, inserting a tab
- `\n` - increments `row` and sets `col` to 0, inserting a new line
- `\b` - decrements `col`, moving the cursor back one
- `0x7f (DEL)` - decrements `col`, prints a blank character, and decrements `col` again, deleting the previous character

4.5 Interrupts and Exceptions

With the ability of visual feedback, the process of debugging becomes much simpler meaning more complex and abstract functionality may be developed. With this in mind, the next important feature implemented is interrupts and exceptions.

4.5.1 Exception Vector Table

In the Raspberry Pi, when an exception occurs, a specific address is loaded into the program counter and execution branches to this point in the code. Therefore, special handlers need to be written at these locations in order for the kernel to branch to the correct exception-handling routine. This set of

Address	Exception type	Meaning
`0x00`	Reset	Hardware reset
`0x04`	Undefined	Executing a garbage instruction
`0x08`	Software Interrupt (SWI)	Software wants to execute a privileged instruction
`0x0c`	Prefetch Abort	Bad memory access of instruction
`0x10`	Data Abort	Bad memory access of data
`0x14`	Reserved	–
`0x18`	Interrupt Request (IRQ)	Hardware signal sent to the CPU
`0x1c`	Fast Interrupt Request (FIQ)	Hardware signal that must be dealt with quickly[6]

Table 3: The Exception Vector Table

addresses is known as the Exception Vector Table, and begins at address `0x0`. The layout of the table is as follows [15, pg. A2-16]:

4.5.2 Processor Modes

When an exception occurs, the CPU switches from whatever mode it was in to a special interrupt mode for the type of exception which was triggered (for example, Abort mode, IRQ mode, FIQ mode). Each of these modes have their own set of alternative mode-specific registers mapped to `r13` and `r14` [16, pg.2-19], allowing for a private stack pointer and link register for each mode. Each mode also contains its own banked version of the SPSR, not available in the standard User and System modes. Since, during an exception, we are interrupting code which did not consent to calling a function, we must save all of the registers it was using, in order that we can eventually return to its execution with it effectively knowing nothing of the interrupt. Therefore, when an exception is triggered we first switch to Supervisor mode (mode `0x13`) and disable interrupts. We then save all of the registers of the previously-executing function, so that we may return control to it later; we then call the exception handler (see Section 4.5.4), and once this terminates we restore the saved registers, and return to the address stored in the link register, continuing execution of the function that was interrupted. This is detailed in Listing 5.

```
irq_handler_wrapper:
    sub lr, lr, #4
    srsdb sp!, #0x13       // Store Return State
    cpsid if, #0x13        // disable IRQs and FIQs in Supervisor mode
    push {r0-r3, r12, lr} // save function's registers
    and r1, sp, #4
    sub sp, sp, r1         // aligns stack to 8-bytes
    push {r1}
    bl irq_handler         // call exception handling routine
    pop {r1}
    add sp, sp, r1         // restores stack alignment
    pop {r0-r3, r12, lr}  // restore saved registers
    rfeia sp!              // Return From Exception
```

Listing 5: The setup and cleanup for dealing with an exception

4.5.3 Function attributes

Important to note is that exception handlers are not regular functions – certain things must be dealt with before and after a regular function is executed which we may not assume is the case when using an exception handling routine. GCC provides function attributes in order to specify certain properties that may help the compiler to optimise certain calls or perform additional checks for correctness. In particular, we may use the ARM `interrupt` attribute to indicate that a function is an interrupt handler, which the compiler uses to generate the suitable function entry and exit sequences. Moreover, use of an

[6]An FIQ takes priority over an IRQ, and only one FIQ source at a time is supported, which reduces latency as the source of the interrupt need not be determined. Furthermore, as an FIQ has its own set of banked registers a state save is not required, reducing the context switch overhead [17].

optional parameter can be used to specify the type of interrupt, and can be one of `IRQ`, `FIQ`, `SWI`, `ABORT`, and `UNDEF` [32]. As an example, the reset handler has the following function prototype:

```
void __attribute__((interrupt("ABORT"))) reset_handler(void);
```

Listing 6: Reset handler prototype

Now, in order to set up the Exception Vector Table, we must ensure that when an exception occurs, we load the absolute address of the handler. This is done by copying the branch instruction from the `.text` section to address `0x0` at runtime – we require this extra step (as opposed to simply loading the branch instruction into the program counter) to ensure we load the Exception Vector Table at the absolute address `0x0`, and that we do not load it relative to the program counter's current position.

The code for dealing with an exception is found in `vector_table.S`. When an exception is raised, we save registers 4 to 9 by pushing them to the stack, as these are the ones that the executing function will have been using (see Section 3.2.3). Next, we copy the 8 exception words from their starting location into registers, then write the register values to address `0x0` – this copies each handler's absolute address to `0x0`, so that the correct handling function may be branched to. After this, we pop registers 4 to 9 and control returns to the function which raised the exception.

4.5.4 Interrupt Requests

An IRQ is a notification to the CPU that something has occurred within the hardware that it needs to be aware of, for example, a keypress, mouse movement, or receiving data from a modem or network card. In order to determine which hardware devices can trigger interrupts, and which may have triggered one, we make use of the Interrupt Controller peripheral, located at offset `0xb000` from the Peripheral Base Address. This peripheral contains three types of registers: pending, which indicates whether a given interrupt has been triggered and used to determine which device has triggered an IRQ exception; enable, which can enable certain interrupts to be triggered; and disable, which can disable the triggering of certain interrupts. Since the Interrupt Controller is a peripheral is a peripheral just like any other, it contains registers which may be communicated with via MMIO, given in Table 4. These detail, for example, whether an IRQ is pending and control if the IRQ is enabled or disabled for a given interrupting device [19, pg. 112]. The reason for the multiple registers which would seemingly do the same thing is that the Broadcom chip can generate two types of interrupt: those coming from the GPU peripherals, and those coming from local ARM peripherals. These "duplicate" registers simply allow for these different sources to be handled accordingly.

Offset	**Register**
`0x200`	IRQ basic pending
`0x204`	IRQ pending 1
`0x208`	IRQ pending 2
`0x20c`	FIQ control
`0x210`	Enable IRQs 1
`0x214`	Enable IRQs 2
`0x218`	Enable basic IRQs
`0x21c`	Disable IRQs 1
`0x220`	Disable IRQs 2
`0x224`	Disable basic IRQs

Table 4: Registers on the Interrupt Controller peripheral

Overall, the Pi has 72 different IRQs, most of which are shared by the ARM CPU and the VideoCore GPU (IRQ 0-63), while some are specific to the CPU (IRQ 64-71). In order to use the Interrupt Controller, we must know the mapping between IRQs and devices; such a mapping is given in Table 5 [33].

4.5.5 Handling an IRQ

For each IRQ we wish to handle, we must write a specific handling routine, for which we define the type `interrupt_handler`. We then declare a static array to hold each handler. The reference manual

IRQ number	Device
0	System Timer Compare Register 0
1	System Timer Compare Register 1
2	System Timer Compare Register 2
3	System Timer Compare Register 3
9	USB Controller
55	PCM Audio
62	SD Host Controller

Table 5: IRQs on the BCM2835

also states [19, pg. 109] that each interrupt-triggering source has a read/write enable bit, specifying whether interrupts are enabled for this device, and a read-only pending bit, to tell us if the device has actually triggered an interrupt. Thus, the pending bit may not be cleared using the Interrupt Controller to overwrite the bit – instead this it must be cleared using the hardware device which initially triggered the interrupt. We therefore define the function type `interrupt_clearer` which, for a given device IRQ, modifies the device's registers in order to deal with the clearing the interrupt. For example, with the System Timer, we set the `matched1` register to 1 (see Section 4.6), and do not touch the Interrupt Controller pending bit itself. We then register the handler by making note of its IRQ number and set the function callbacks for its handler and clearer functions, using the function:

```
void register_irq_handler(
        enum irq_no num,
        interrupt_handler handler,
        interrupt_clearer clearer
    );
```

Listing 7: Registering an IRQ

Each time an interrupt is triggered, we iterate over each of the 72 possible interrupts and use the Interrupt Controller to check the bit specifying whether this IRQ has been triggered (i.e. the IRQ is pending). If this is the case, we call the `clearer()` function to signal that we have acknowledged the pending interrupt, and then execute its `handler()` function.

4.5.6 Initialising Interrupts

We begin by zeroing the entirety of the interrupt handler and interrupt clearer functions, and write `0xffffffff` to set all bits in each disable register in the Interrupt Controller. We then perform the copying of the Exception Vector to address, thus completing initialisation.

It is important to be able to enable and disable interrupts entirely, for example to avoid being interrupted before we have dealt with another IRQ. This is done using the `cps`, or Change Program State, instruction. We wish to change the bit associated with IRQs, denoted by `i` or bit 7 in the CPSR [16, pg. 2-11], meaning to enable interrupts we may simply call:

```
cpsie i
```

Listing 8: Enabling and disabling interrupts

Disabling interrupts is achieved by replacing the suffix `ie` (Interrupts Enable) with `id` (Interrupts Disable). Note that we may simply query the state of interrupts by checking bit 7 of the CPSR. This is done by storing the CPSR into a register using `mrs`, followed by an arithmetic shift right by 7 places to access the seventh bit, and performing a logical AND to determine its value. If it is clear, then interrupts are enabled:

```
mrs r0, cpsr
asr r0, r0, #7
and r0, r0, #1
```

Listing 9: Checking the status of interrupts

4.6 System Timer

With the ability to generate and deal with interrupts sent to the CPU, we now focus on implementing the system timer, a peripheral which can keep time and send interrupts after a set amount of time. This is located at offset `0x3000` from the Peripheral Base Address, and contains only the following seven registers [19, pg. 172]:

Offset	**Register**
`0x00`	Control/Status
`0x04`	Counter (lower 32 bits)
`0x08`	Counter (upper 32 bits)
`0x0c`	Compare Register 0
`0x10`	Compare Register 1
`0x14`	Compare Register 2
`0x18`	Compare Register 3

Table 6: Registers on the System Timer peripheral

The timer operates by incrementing a 64-bit counter, made up of two separate 32-bit registers, every microsecond. It starts as soon as the system boots, and runs constantly in the background as long as the Pi is switched on. There are four registers with which the system timer will compare the lower 32 bits of the counter each tick, and each is capable of triggering an IRQ – if any of these registers match the counter, an IRQ is generated from the register that matched. As shown in Table 5, the IRQ numbers for these registers are 0-3; 0 and 2 are used by the GPU and must not be touched, leaving registers 1 and 4 available to use.

The Control/Status Register contains flags in its least-significant four bits which indicate whether an interrupt has been triggered, with a bit in each place signifying that this register has triggered an IRQ [19, pg. 173]. To access these bits, we define a structure that makes use of bit-fields so that we may easily read and write these flags. As touched upon in 4.5.5, to clear the IRQ for the system timer we write a 0 to the bit representing the appropriate register that has been matched by the counter. Simply modifying the Interrupt Controller's registers directly will result in not clearing the IRQ. The overall structure for the system timer is given in Listing 10, while the Control/Status Register is given in Listing 11.

```
/* system timer peripheral register */
struct sys_timer {
    struct timer_ctrl control;
    uint32_t counter_low;
    uint32_t counter_high;
    uint32_t compare0;
    uint32_t compare1;
    uint32_t compare2;
    uint32_t compare3;
};
```

Listing 10: The System Timer

```
/* system timer control/status register */
struct timer_ctrl {
    uint8_t matched0 : 1;
    uint8_t matched1 : 1;
    uint8_t matched2 : 1;
    uint8_t matched3 : 1;
    uint32_t reserved : 28;
};
```

Listing 11: The Control/Status Register

The final piece of functionality to implement for the system timer is the ability to set the timer to go off in a number of microseconds. Recall that compare registers 0 and 2 are used by the GPU and hence not available for general use, meaning we may use registers 1 and 3; the project uses the former, but it makes no difference which is chosen. The timer is set by setting the `compare1` value to the current "tick" plus some number of microseconds – this value is compared each microsecond by the system timer, and when the timer's counter and this register match, an IRQ is sent by the `matched1` register. This functionality becomes particularly useful in process scheduling, which is covered in Section 4.8.2. We also use this to implement `uwait(usecs)`, analogous to the C standard library's `udelay()` – while the difference between the current ticks and the ticks on the counter when the call to `uwait()` was made is less than the microseconds parameter `usecs`, do nothing. This effectively waits for `usecs` microseconds.

As a final note on the timer, recall from Table 6 that the system timer is effectively a 64-bit counter, making use of two 32-bit counters to represent the upper and lower 32 bits. The project only makes use of the latter, making the assumption that the system will be running for less than $2^{32}\mu s \approx 72$ minutes. This is perhaps too restrictive for a real-world operating system, but attention can be focused on this issue at a later date – there are more significant problems which currently affect its widespread adoption. Moreover, simply using the lower counter illustrates the peripheral's use adequately enough.

4.7 Processes

4.7.1 Process Control Blocks

We begin with our implementation of processes by defining for the operating system what exactly a process looks like – this may differ from system to system, but in general will contain information such as the contents of the CPU registers, the process' state (be it running, ready, waiting, etc.), a unique identifier, a program counter, containing the address of the next instruction to be executed, and any CPU scheduling information. The representation of a process in an operating system is known as a Process Control Block (PCB) [7, pg. 105]. In this project, since there is currently little that they can do, processes are only defined by a few fields, namely:

```c
/* process control block */
struct proc {
    struct cpu_state *state;
    uint32_t pid;
    char name[32];
    void *stack_page;
    DEFINE_LINK(proc);
};
```

Listing 12: A Process Control Block

Here, `struct proc_state` simply contains data about the contents of the CPU registers (`r0-r15`); `pid` is an identifying number (for the operating system's use); `name` is an identifying string (for human use); `stack_page` is the address of the page which the process uses; and `DEFINE_LINK(proc)` is a macro declaring pointers to other processes, `next` and `prev`, for the various process queues used later.

There are two vital pieces of functionality for processes to be of any use – creating a new process and bringing it into memory for execution, and cleaning up after it has terminated. To this end we implement two functions, `create_kthread()` and `cleanup()`.

4.7.2 Creating a process

A process is created by defining each of the fields in Listing 12. Each thread gets its own new page using the `alloc_page()` function from Section 4.2.2, while the CPU state (the copy of each of the CPU registers) is zeroed and located at the last 60 bytes of this page. We then set three fields within the CPU state structure: the link register, the stack pointer, and the CPSR.

The link register contains the address to which to jump when a function call returns, and will therefore hold the address of the process' main function, which is passed as a parameter to the function. We define the new data type `kthreadfn` for this purpose. The stack pointer, meanwhile, contains the address of the function `cleanup`, which is responsible for clearing up after the process terminates. The CPSR is set to `0x13`, the code for Supervisor mode, as all threads currently run in kernel-space and therefore have no restrictions. Each time we create a new thread, we also add it to the job queue and ready queue, for use in process scheduling. Therefore, all the hobbyist programmer must do to create a new process is specify the function it will be executing and assign it a name to keep track of it in the system.

4.7.3 Reapers

Since we want to run multiple different processes throughout the execution of the operating system, it would not make sense to keep once which have finished executing in memory, taking up space that new ones could use. Therefore, we define the `cleanup()` function that frees all the memory associated with the process, removes it from both the job queue and the ready queue, and allows the next process in the ready queue to use the CPU.

4.7.4 Initialising Processes

To initialise processes we create an “init” process, `init`, just as we would create any other process, then add it to both the job queue and ready queue. Since at the time of its creation it is the only process in the system, it will continue to execute and initialise the rest of the processes. We also define a variable to specify the currently executing process and set it to `init`, then set the system timer to go off in a set amount of time (a *quantum*) to start off process scheduling.

4.8 Scheduling

During execution, regular processes have no consideration for other processes – if they had their way they would use the CPU and keep it to themselves until they had reached the end of their task. We therefore implement an interface to allow for different process scheduling policies to be followed, which follow different approaches to systematically passing control of the CPU from one process to the next, and provide the ability to configure this functionality at compile-time by passing command-line directives to the Makefile.

4.8.1 Context Switching

Before we may write a scheduling policy by which our system will adhere, we must define how the use of the CPU will be passed from one executing process to another. Similarly to how interrupts are dealt with, in order that we may eventually continue execution of the process we are taking off of the CPU, we must save all of the registers in use by the process onto the stack, and importantly we save its stack pointer so that execution may resume from where it left off when control of the CPU is eventually returned to the process. A context switch comprises of two actions: a state save, where the process is stored in memory; and a state restore, where a previously saved process is loaded back into memory. This is illustrated in Listing 13.

```
.equ QUANTUM, 20000
switch_context:
    /* save current process' state */
    push {lr}
    push {sp}
    mrs r12, cpsr     /* get current status register */
    push {r0-r12}     /* save general purpose regs and state */
    str sp, [r0]      /* store stack pointer */

    /* load new process' state */
    ldr sp, [r1]
    ldr r0, =QUANTUM      /* quantum time of 20ms */
    bl timer_set          /* set timer to go off in one quantum */
    pop {r0-r12}
    msr cpsr_c, r12
    pop {lr, pc}
```

Listing 13: Context switch

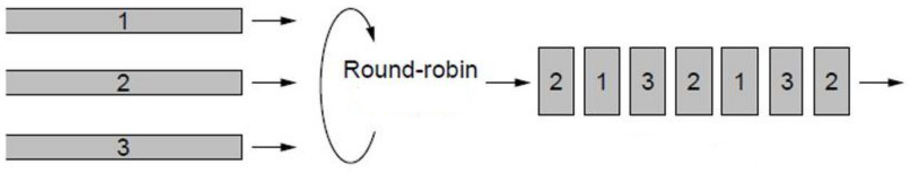

Figure 3: The Round Robin scheduler

4.8.2 Round Robin

The first process scheduling policy we implement is the Round Robin scheduler [7, pg. 271]. This sequentially gives each process in the ready queue uninterrupted use of the CPU for one quantum, after which use of the CPU is passed to the next process in the queue, regardless of the process' progress. This policy is illustrated in Figure 3. The implementation itself is rather simple: we first disable interrupts, in order to ensure the successful completion of the context switch, then test whether there is another job in the ready queue. If not, there will be nothing to switch context to, so we set the system timer to go off in another quantum, which we set as $20\mu s$, and re-enable interrupts. If there are other jobs, however, we add the current one to the end of the ready queue and switch context to the process that was at the head. We then finish by re-enabling interrupts.

4.8.3 First Come First Served

The project also implements the First Come First Served process scheduler – in contrast to the Round Robin scheduler, this gives a process use of the CPU until it no longer needs it. While this seems less "fair" (precise scheduling criteria can be found at [7, pg. 265]), this scheduling policy may find use if it is vital that processes are executed in the order they are received. This algorithm is simple and provided largely to illustrate the use of compile-time options to configure the system, one of the key goals of the project.

4.8.4 Extension

With ease of extension a key consideration of this project, especially for a task such as process scheduling, care was taken in designing the interface for adding new scheduling algorithms. In particular, as it stands, all the code required for configuring the scheduling policy to use is contained in the file `sched.h` – all that is required to add a policy to the system is to a) write the function that will be passing control to and taking control from the CPU, and b) add a case in the include guards to add this new scheduler as an option, so that the correct scheduler is used. A global function pointer, `void (*schedule)(void)`, taking no arguments and returning nothing, is used to set the scheduling policy of the system, and is set by testing which command-line options have been passed by the Makefile. The interface is covered more in depth in Section 4.10.

Overall this interface works well – it allows for the different schedulers to be switched between easily and with little effort. Its ease of extension could, however, face challenges as the policies the schedulers adopt become more complex. The scheduler the system uses is innately tied to influences how a process is represented in that system; for example, if one wanted to implement priority scheduling [7, pg. 210], the structure `struct proc` would need to be extended to include how important it is that a given process is executed, with potentially a field to describe how this may change over time (aging). Therefore, changes will need to be made to at least the files `proc.h`, `sched.h`, and `sched.c`. This could eventually become cumbersome, and is not a consideration that the project takes in its current form. Furthermore, when using the First Come First Served scheduler, it does not disable the triggering of interrupts after a quantum has passed, but rather just ignores them. While this works, it is perhaps conceptually bad practise, and could lead to confusion, which this project aims to explicitly avoid. While there has not been the time or urgency to focus on this so far, it will certainly be a consideration in the future.

4.9 Synchronisation

Since the project is a concurrent system, there is the possibility for race conditions – the outcome of a computation being affected by the order in which processes access data. For this, we implement two forms of locks: spinlocks and semaphores. Both rely on the concept of an atomic operation, which is a computation that is performed with the guarantee of not being interrupted. For this purpose, ARM provides the `swp` instruction [17], essentially allowing us to "acquire" the lock first and then check that we have been successful, as opposed to the other way around. The implementation of a such an operation is given in Listing 14.

```
/* int try_lock(int *) */
try_lock:
    mov r1, #0
    swp r2, r1, [r0]
    mov r0, r2
    blx lr
```

Listing 14: Atomic lock operation

We use this to implement the most basic form of synchronisation, the spinlock. This involves constantly trying to acquire the lock until it is successful. We initialise the lock value to 1, and the lock only successfully acquires it if it finds the result of `try_lock()` is 1.

More useful is a semaphore, and in particular the project implements the design outlined in [7, pg. 265]. To reduce the wastage of CPU cycles caused by busy-waiting that spinlocks entail, we instead maintain a list of processes that wish to acquire the semaphore: each process in this queue will be placed in the waiting state, meaning that the CPU scheduler can transfer control of the CPU to another process that is in the ready queue. This greatly increases CPU usage as control is always given to a process that can perform meaningful work with it, rather that simply waiting for the lock variable. Later, when a process releases the lock, it uses this wait queue to give the lock to the next process that was waiting for it.

The implementation of this synchronisation interface takes inspiration from the POSIX mutex library [34], albeit far more simplified, requiring only one single-parameter function call to initialise a given lock. When `mutex_lock()` is called, a process tries acquire the lock and is either given the lock or placed on the wait queue, while `mutex_unlock()` simply gives the lock to the next process in the queue.

4.10 Configuration

As mentioned, one of the key aims of the project was to present both an extensible and configurable operating system in order to demystify certain aspects about their development. While being clear to read and understand is useful, the ability to actively change key systems at compile-time provides a much more practical and hands-on opportunity to understand the key concepts that underpin the system. The main aspect to system configuration in this project is the use of Makefile directives, which provides an intuitive interface to quickly compile the system to make use of different approaches, and while not used extensively so far, it is an interface whose simplicity to extend to more options in the future is helpful.

4.10.1 Makefile

Compile-time configuration is performed by sending command-line directives to the Makefile. In particular, the project makes use of the `-D` flag to define certain variables within the source code. For example, when the target model was switched to the Pi 1 Model B+ from the Pi 2 Model B as a result of difficulties with the framebuffer, the code that dealt with shutting down three of the four cores was simply wrapped in C preprocessor `#if defined` tests, meaning it would only get executed if it is explicitly specified that we are compiling for the Raspberry Pi 2 Model B. If not, it is assumed that we compile for the Raspberry Pi 1. This solution was particularly useful as it makes the codebase more portable to other models, without breaking its functionality on the current model and without requiring to delete the code entirely.

Even more useful, considering the initial goals of the project, is the ability to load different process schedulers at compile-time. This is achieved in much the same way as how the model of Raspberry Pi is targeted – we simply wrap the blocks of code corresponding to the different options available in preprocessor expressions, and set the `void (*schedule)(void)` function accordingly. Listing 15 gives an example issue of the `make` command with command-line directives sent to the preprocessor, and shows which directives they define. Listing 16 shows how this interface is used to determine which scheduling function to compile. Just as the model requires a default value, we assume the default scheduler to use is the Round Robin algorithm.

```
make model=1 sched=robin
DIRECTIVES = -DRPIBPLUS -DSCHED_ROUNDROBIN
```

Listing 15: An example `make` command, and the directives it defines

```
#if defined ( SCHED_FCFS )
    schedule = sched_fcfs;
    strcpy(sch_str, "FCFS");
#elif defined ( SCHED_ROUNDROBIN )
    schedule = sched_round_robin;
    strcpy(sch_str, "Round Robin");
#else
    schedule = sched_round_robin;
    strcpy(sch_str, "Round Robin");
#endif
```

Listing 16: Configuring the scheduling algorithm at compile-time

4.11 Ongoing development

The project, while functional in what it implements, is not in an entirely complete state. Two systems in particular have been developed to an extent, but work is required to integrate them fully into the operating system – these are the tasks of inter-process communication and user interaction via a keyboard.

4.11.1 Inter-process Communication

The project implements a functional but rudimentary form of the shared-memory model of inter-process communication. This involves declaring a new structure, `struct shm_section`, which contains fields describing its address in memory, an identifying name, and a character array to storo data written to it. We create a new shared memory section by allocating a new page, with the `shared` bit set. Another process may then use the data in this page with calls to `shm_read()` and `shm_write()`, which simply perform MMIO with extra checks for validity of access.

4.11.2 Keyboard input

Keyboard input is generally done via USB keyboard on the Raspberry Pi, although we may use the UART peripheral to receive input from a keyboard using a USB-to-TTL adapter. Since the USB standard is designed to accommodate a wide variety of hardware devices, with USB ports themselves containing only 4 pins, this means much of the functionality must be implemented in hardware. However, the time frame of the project did not allow for such a task as implementing a USB keyboard driver from scratch. Instead, it was opted to use a static library to provide the functionality of user interaction. In particular, Rene Stange provides such a driver [24], whose compilation may be tweaked depending on the target environment. This is then archived into a static library, whose functions may be accessed by supplying it to the linker at compile-time. In the case of this project, this is done by specifying `-luspi` as a linker flag in the Makefile. However, this solution still too remains in the process of being developed to be entirely functional.

5 Testing and Issues

5.1 Loading onto real hardware

Vital to ensuring that the system truly works is testing it on real hardware. As covered in Section 3.1.1, the Pi requires firmware containing its three bootloaders in order to bring a kernel image into memory and begin execution. The most simple way to acquire this was to download an existing operating system for the Pi, since this would already contain all the files required, and simply replace the `kernel.img` with that generated during compilation. We use the toolchain utility `arm-none-eabi-objcopy` to copy the generated ELF file to a raw binary image. The image is then copied to the SD card to replace that of Raspbian, meaning the project's operating system will run in its place.

5.1.1 Integer division on the ARM CPU

When compiling, GCC raised the error of an unrecognised symbol within the compiled object files, `__aeabi_uidivmod`, and after debugging it was discovered to have originated from use of the modulo operator. The cause of this is the fact that the ARM family of CPUs does not implement a native integer division instruction [35], and thus requires a custom implementation. A solution, at least early on, was to link against the `-lgcc` flag, as GCC provides its own implementation in this library. However, since an important, if implicit, personal requirement of the project is that it is self-sufficient wherever possible, it was decided to write the long division algorithm in assembly. The `__aeabi_*div*` family of functions require that the quotient of the division is stored in `r0` by the time the function returns, and that the remainder is in `r1`. Listing 17 shows the implementation of `__aeabi_divmod(numerator, denominator)`, which allows for unsigned integer division and modulo operations. The implementation for `__aeabi_uidiv()` simply calls `__aeabi_uidivmod()`, as both have the same requirements regarding the contents of registers 0 and 1.

```
__aeabi_uidivmod:
    mov r2, #0  // quotient
    loop:
        cmp r0, r1
        blo return
        sub r0, r0, r1
        add r2, r2, #1
        b loop
    return:
        mov r1, r0
        mov r0, r2
        bx lr
```

Listing 17: Implementation of unsigned integer division and modulo in ARM assembly

5.2 Interacting with the Memory Management Unit

Another issue during testing was the compiler warning of the deprecation of the `swp` instruction in ARMv6 and higher, as used in the implementation of the atomic swap operation in Listing 14. It was discovered that the Load Register Exclusive and Store Register Exclusive instructions, `ldrex` and `strex` respectively, were preferred for this operation. It was further discovered [36] that these instructions were available with both caches and the MMU enabled. While the project does not work when using these exclusive loads and stores, causing a data abort when attempting to do so, it does succeed in initialising the MMU, allowing the programmer to define their own map from virtual to physical memory using the `mmu_section()` interface. Much of this code is, however, accredited to David Welch, having been largely lifted from [37]. Not being a primary objective of the project, however, and having been initially with the intention simply to use these non-deprecated instructions, the use of this code was judged to be permissible.

5.3 ACT debugging

Lastly, the most notable and rewarding phase of testing was performed via the GPIO – after loading onto real hardware, but before getting framebuffer initialisation working, there was no means of getting visual feedback from the real-world (i.e. non-emulated) system, other than by blinking the ACT LED.

6 Evaluation

6.1 Achievements

6.2 Limitations

6.3 Further work

References

[1] "Downloads." https://www.raspberrypi.org/downloads/. Page accessed: 2018-11-22.

[2] "Baking pi - operating systems development." https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/index.html. Page accessed: 2018-11-22.

[3] "Building an operating system for the raspberry pi." https://jsandler18.github.io/. Page accessed: 2018-05-17.

[4] B. Pfaff, "Pintos." https://web.stanford.edu/class/cs140/projects/pintos/pintos.html#SEC_Top, December 2009. Document accessed: 2018-08-24.

[5] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation.* Pearson, 3 ed.

[6] E. Helin and A. Renberg, "The little book about os development." https://littleosbook.github.io/, January 2015. Document accessed: 2018-11-22.

[7] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts.* Wiley, 9 ed., 2014.

[8] "Raspberry pi documentation: Bcm2836." https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/README.md. Page accessed: 2018-10-08.

[9] "Understanding the raspberry pi boot process." https://wiki.beyondlogic.org/index.php?title=Understanding_RaspberryPi_Boot_Process. Page accessed: 2019-04-13.

[10] "Github: raspberrypi/firmware." https://github.com/raspberrypi/firmware. Page accessed: 2019-04-13.

[11] "Raspberry pi: Level of hackability of raspberry pi." https://raspberrypi.stackexchange.com/questions/7122/level-of-hackability-of-raspberry-pi/7126#7126. Page accessed: 2018-11-15.

[12] "Raspberry pi: What bios does raspberry pi use?." https://raspberrypi.stackexchange.com/questions/8475/what-bios-does-raspberry-pi-use. Page accessed: 2018-11-15.

[13] "Gnu-rm downloads." https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads. Page accessed: 2018-11-18.

[14] "arm-none-eabi-gcc 8.2.0-1." https://www.archlinux.org/packages/community/x86_64/arm-none-eabi-gcc/. Page accessed: 2018-11-18.

[15] ARM, *ARM Architecture Reference Manual*, 2005. Issue I.

[16] ARM, *ARM1176JZF-S Technical Reference Manual*, 2009. Revision r0p7.

[17] "Arm developer suite." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.ads/index.html. Page accessed: 2019-04-20.

[18] ARM, *ARM Cortex-A Series: Programmer's Guide*, 2013.

[19] Broadcom Corporation, Broadcom Europe Ltd., 406 Science Park, Milton Road, Cambridge, *Broadcom BCM 2835 ARM Peripherals*, 2012.

[20] G. van Loo, *Broadcom BCM 2836 ARM Peripherals.* Broadcom Corporation, Broadcom Europe Ltd., 406 Science Park, Milton Road, Cambridge, 2014.

[21] "C: Things c can't do." https://wiki.osdev.org/C#Things_C_can.27t_do. Page accessed: 2018-08-30.

[22] "Gnu make: Writing recipes in rules." https://www.gnu.org/software/make/manual/make.html#toc-Writing-Recipes-in-Rules. Page accessed: 2019-04-20.

[23] "Make: How to use variables." https://ftp.gnu.org/old-gnu/Manuals/make-3.79.1/html_chapter/make_6.html. Page accessed: 2019-04-20.

[24] "rsta2: A bare metal usb driver for raspberry pi written in c." https://github.com/rsta2/uspi. Page accessed: 2019-04-22.

[25] "Doxygen." http://doxygen.nl/. Page accessed: 2019-04-24.

[26] "Javadoc tool." https://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html. Page accessed: 2019-04-24.

[27] "nm(1) - linux man page." https://linux.die.net/man/1/nm. Page accessed: 2019-04-24.

[28] "Trying bare metal on raspberry pi 2." https://www.raspberrypi.org/forums/viewtopic.php?p=693724. Page accessed: 2019-04-24.

[29] "Booting arm linux: Tag reference." http://www.simtec.co.uk/products/SWLINUX/files/booting_article.html#appendix_tag_reference. Page accessed: 2018-11-20.

[30] "Raspberry pi firmware: Mailboxes." https://github.com/raspberrypi/firmware/wiki/Mailboxes. Page accessed: 2018-11-21.

[31] "dhepper/font8x8: 8x8 monochrome bitmap fonts for rendering." https://github.com/dhepper/font8x8/blob/master/font8x8_basic.h. Page accessed: 2019-04-26.

[32] "Arm function attributes." https://gcc.gnu.org/onlinedocs/gcc/ARM-Function-Attributes.html#ARM-Function-Attributes. Page accessed: 2019-04-26.

[33] "Bcm2835 interrupt controller." https://embedded-xinu.readthedocs.io/en/latest/arm/rpi/BCM2835-Interrupt-Controller.html. Page accessed: 2019-04-26.

[34] "pthread mutex lock(3) - linux man page." https://linux.die.net/man/3/pthread_mutex_lock. Page accessed: 2019-04-28.

[35] "Divide and conquer." https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/divide-and-conquer. Page accessed: 2019-03-21.

[36] https://www.raspberrypi.org/forums/viewtopic.php?p=1204251. Page accessed: 2019-03-27.

[37] "dwelch67/raspberrypi/mmu." https://github.com/dwelch67/raspberrypi/tree/master/mmu. Page accessed: 2019-04-03.

Appendix A Timetable